W3School Redis 教程 & 命令参考

wizardforcel

Published with GitBook



目錄

W3School Redis 教程 & 命令参考

来源：

- Redis 教程
- Redis 命令参考

整理：飞龙

Redis 教程

Redis 简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

- Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
- Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
- Redis支持数据的备份，即master-slave模式的数据备份。

Redis 优势

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

- Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
- Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，应为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

Redis 安装

Window 下安装

下载地址：https://github.com/dmajkic/redis/downloads。

下载到的Redis支持32bit和64bit。根据自己实际情况选择，将64bit的内容cp到自定义盘符安装目录取名redis。 如 C:\reids

打开一个cmd窗口 使用cd命令切换目录到 C:\redis 运行 **redis-server.exe redis.conf** 。

如果想方便的话，可以把redis的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个redis.conf可以省略，如果省略，会启用默认的。输入之后，会显示如下界面：

```
C:\WINDOWS\system32\cmd.exe - redis-server.exe redis.conf
C:\redis>redis-server.exe redis.conf
[7468] 13 Sep 14:36:13 * Server started, Redis version 2.4.5
[7468] 13 Sep 14:36:13 * DB loaded from disk: 0 seconds
[7468] 13 Sep 14:36:13 * The server is now ready to accept connections on port 6
379
[7468] 13 Sep 14:36:14 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:14 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:20 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:20 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:25 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:25 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:31 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:31 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:36 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:36 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:42 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:42 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:47 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:47 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:52 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:52 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:36:58 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:36:58 - 0 clients connected (0 slaves), 672860 bytes in use
[7468] 13 Sep 14:37:03 - DB 0: 1 keys (0 volatile) in 4 slots HT.
[7468] 13 Sep 14:37:03 - 0 clients connected (0 slaves), 672860 bytes in use
```

这时候别启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。

切换到redis目录下运行 **redis-cli.exe -h 127.0.0.1 -p 6379** 。

设置键值对 **set myKey abc**

取出键值对 **get myKey**

```
C:\WINDOWS\system32\cmd.exe - redis-cli.exe -h 127.0.0.1 -p 6379
C:\redis>redis-cli.exe -h 127.0.0.1 -p 6379
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> set myKey abc
OK
redis 127.0.0.1:6379> get myKey
"abc"
redis 127.0.0.1:6379>
```

Linux 下安装

下载地址：http://redis.io/download，下载最新文档版本。

本教程使用的最新文档版本为 2.8.17，下载并安装：

```
$ wget http://download.redis.io/releases/redis-2.8.17.tar.gz
$ tar xzf redis-2.8.17.tar.gz
$ cd redis-2.8.17
$ make
```

make完后 redis-2.8.17目录下会出现编译后的redis服务程序redis-server,还有用于测试的客户端程序redis-cli

下面启动redis服务.

```
$ ./redis-server
```

注意这种方式启动redis 使用的是默认配置。也可以通过启动参数告诉redis使用指定配置文件使用下面命令启动。

```
$ ./redis-server redis.conf
```

redis.conf是一个默认的配置文件。我们可以根据需要使用自己的配置文件。

启动redis服务进程后，就可以使用测试客户端程序redis-cli和redis服务交互了。 比如：

```
$ ./redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

Ubuntu 下安装

在 Ubuntu 系统安装 Redi 可以使用以下命令:

```
$sudo apt-get update
$sudo apt-get install redis-server
```

启动 Redis

```
$redis-server
```

查看 redis 是否启动？

```
$redis-cli
```

以上命令将打开以下终端：

```
redis 127.0.0.1:6379>
```

127.0.0.1 是本机 IP ，6379 是 redis 服务端口。现在我们输入 PING 命令。

```
redis 127.0.0.1:6379> ping
PONG
```

以上说明我们已经成功安装了redis。

Redis 配置

Redis 的配置文件位于 Redis 安装目录下，文件名为 redis.conf。

你可以通过 **CONFIG** 命令查看或设置配置项。

语法

Redis CONFIG 命令格式如下：

```
redis 127.0.0.1:6379&gt; CONFIG GET CONFIG_SETTING_NAME
```

实例

```
redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"
2) "notice"
```

使用 * 号获取所有配置项：

实例

```
redis 127.0.0.1:6379> CONFIG GET *

  1) "dbfilename"
  2) "dump.rdb"
  3) "requirepass"
  4) ""
  5) "masterauth"
  6) ""
  7) "unixsocket"
  8) ""
  9) "logfile"
 10) ""
 11) "pidfile"
 12) "/var/run/redis.pid"
 13) "maxmemory"
 14) "0"
 15) "maxmemory-samples"
 16) "3"
 17) "timeout"
 18) "0"
```

```
19) "tcp-keepalive"
20) "0"
21) "auto-aof-rewrite-percentage"
22) "100"
23) "auto-aof-rewrite-min-size"
24) "67108864"
25) "hash-max-ziplist-entries"
26) "512"
27) "hash-max-ziplist-value"
28) "64"
29) "list-max-ziplist-entries"
30) "512"
31) "list-max-ziplist-value"
32) "64"
33) "set-max-intset-entries"
34) "512"
35) "zset-max-ziplist-entries"
36) "128"
37) "zset-max-ziplist-value"
38) "64"
39) "hll-sparse-max-bytes"
40) "3000"
41) "lua-time-limit"
42) "5000"
43) "slowlog-log-slower-than"
44) "10000"
45) "latency-monitor-threshold"
46) "0"
47) "slowlog-max-len"
48) "128"
49) "port"
50) "6379"
51) "tcp-backlog"
52) "511"
53) "databases"
54) "16"
55) "repl-ping-slave-period"
56) "10"
57) "repl-timeout"
58) "60"
59) "repl-backlog-size"
60) "1048576"
61) "repl-backlog-ttl"
62) "3600"
63) "maxclients"
64) "4064"
65) "watchdog-period"
66) "0"
67) "slave-priority"
68) "100"
69) "min-slaves-to-write"
70) "0"
71) "min-slaves-max-lag"
```

```
 72) "10"
 73) "hz"
 74) "10"
 75) "no-appendfsync-on-rewrite"
 76) "no"
 77) "slave-serve-stale-data"
 78) "yes"
 79) "slave-read-only"
 80) "yes"
 81) "stop-writes-on-bgsave-error"
 82) "yes"
 83) "daemonize"
 84) "no"
 85) "rdbcompression"
 86) "yes"
 87) "rdbchecksum"
 88) "yes"
 89) "activerehashing"
 90) "yes"
 91) "repl-disable-tcp-nodelay"
 92) "no"
 93) "aof-rewrite-incremental-fsync"
 94) "yes"
 95) "appendonly"
 96) "no"
 97) "dir"
 98) "/home/deepak/Downloads/redis-2.8.13/src"
 99) "maxmemory-policy"
100) "volatile-lru"
101) "appendfsync"
102) "everysec"
103) "save"
104) "3600 1 300 100 60 10000"
105) "loglevel"
106) "notice"
107) "client-output-buffer-limit"
108) "normal 0 0 0 slave 268435456 67108864 60 pubsub 33554432 8388
109) "unixsocketperm"
110) "0"
111) "slaveof"
112) ""
113) "notify-keyspace-events"
114) ""
115) "bind"
116) ""
```

编辑配置

你可以通过修改 redis.conf 文件或使用 **CONFIG set** 命令来修改配置。

语法

CONFIG SET 命令基本语法：

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VAL
```

实例

```
redis 127.0.0.1:6379> CONFIG SET loglevel "notice"
OK
redis 127.0.0.1:6379> CONFIG GET loglevel

1) "loglevel"
2) "notice"
```

参数说明

redis.conf 配置项说明如下：

1. Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程

```
**daemonize no**
```

2. 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定

```
**pidfile /var/run/redis.pid**
```

3. 指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

```
**port 6379**
```

4. 绑定的主机地址

```
**bind 127.0.0.1**
```

5.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能

```
**timeout 300**
```

6. 指定日志记录级别，Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose

```
**loglevel verbose**
```

7. 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

```
**logfile stdout**
```

8. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

```
**databases 16**
```

9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

```
**save &lt;seconds&gt; &lt;changes&gt;**

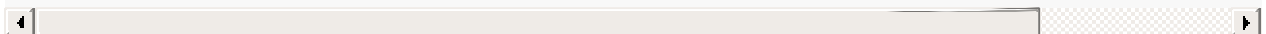
Redis默认配置文件中提供了三个条件：

**save 900 1**
```

save 300 10

save 60 10000

```
分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10
```



10. 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

```
**rdbcompression yes**
```

11. 指定本地数据库文件名，默认值为dump.rdb

```
**dbfilename dump.rdb**
```

12. 指定本地数据库存放目录

```
**dir ./**
```

13. 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

```
**slaveof &lt;masterip&gt; &lt;masterport&gt;**
```

14. 当master服务设置了密码保护时，slav服务连接master的密码

```
**masterauth &lt;master-password&gt;**
```

15. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

```
**requirepass foobared**
```

16. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

```
**maxclients 128**
```

17. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区

```
**maxmemory &lt;bytes&gt;**
```

18. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

```
**appendonly no**
```

19. 指定更新日志文件名，默认为appendonly.aof

```
**appendfilename appendonly.aof**
```

20. 指定更新日志条件，共有3个可选值：
no：表示等操作系统进行数据缓存同步到磁盘（快）
always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）
everysec：表示每秒同步一次（折衷，默认值）

```
**appendfsync everysec**
```

21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）

```
**vm-enabled no**
```

22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

```
**vm-swap-file /tmp/redis.swap**
```

23. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

```
**vm-max-memory 0**
```

24. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值

```
**vm-page-size 32**
```

25. 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。

```
**vm-pages 134217728**
```

26. 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

```
**vm-max-threads 4**
```

27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

```
**glueoutputbuf yes**
```

28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

```
**hash-max-zipmap-entries 64**
```

hash-max-zipmap-value 512

29. 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）

```
**activerehashing yes**
```

30. 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

```
**include /path/to/local.conf**
```

Redis 数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

String（字符串）

string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。

string类型是Redis最基本的数据类型，一个键最大能存储512MB。

实例

```
redis 127.0.0.1:6379> SET name "w3cschool.cc"
OK
redis 127.0.0.1:6379> GET name
"w3cschool.cc"
```

在以上实例中我们使用了 Redis 的 **SET** 和 **GET** 命令。键为 name，对应的值为 w3cschool.cc。

注意：一个键最大能存储512MB。

Hash（哈希）

Redis hash 是一个键值对集合。

Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

实例

```
redis 127.0.0.1:6379> HMSET user:1 username w3cschool.cc password w
OK
redis 127.0.0.1:6379> HGETALL user:1
1) "username"
2) "w3cschool.cc"
3) "password"
4) "w3cschool.cc"
5) "points"
6) "200"
redis 127.0.0.1:6379>
```

以上实例中 hash 数据类型存储了包含用户脚本信息的用户对象。 实例中我们使用了 Redis **HMSET, HEGTALL** 命令，**user:1** 为键值。

每个 hash 可以存储 $2^{32 - 1}$ 键值对（40多亿）。

List（列表）

Redis 的 list 类型其实就是一个每个子元素都是string类型的双向链表。你可以将 元素添加到列表的头和尾。

实例

```
redis 127.0.0.1:6379> lpush w3cschool.cc redis
(integer) 1
redis 127.0.0.1:6379> lpush w3cschool.cc mongodb
(integer) 2
redis 127.0.0.1:6379> lpush w3cschool.cc rabitmq
(integer) 3
redis 127.0.0.1:6379> lrange w3cschool.cc 0 10
1) "rabitmq"
2) "mongodb"
3) "redis"
redis 127.0.0.1:6379>
```

列表最多可存储 2^{32 - 1} 元素 (4294967295, 每个列表可存储40多亿)。

set（集合）

Redis的set是string类型的无序集合。

set的是通过hash table实现的，所以添加， 删除， 查找的复杂度都是O(1)。

sadd 命令

添加一个string元素到,key对应的set集合中，成功返回1,如果元素以及在集合中返回0,key对应的set不存在返回错误。

```
sadd key member
```

实例

```
redis 127.0.0.1:6379> sadd w3cschool.cc redis
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc mongodb
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc rabitmq
(integer) 1
redis 127.0.0.1:6379> sadd w3cschool.cc rabitmq
(integer) 0
redis 127.0.0.1:6379> smembers w3cschool.cc

1) "rabitmq"
2) "mongodb"
3) "redis"
```

注意：以上实例中 rabitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

集合中最大的成员数为 2^{32 - 1} (4294967295, 每个集合可存储40多亿个成员)。

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的score。redis正是通过分数来为集合中的成员进行从小到大的排序。

zset的成员是唯一的,但分数(score)却可以重复。

zadd 命令

添加元素到集合，元素在集合中存在则更新对应score

```
zadd key score member
```

实例

```
redis 127.0.0.1:6379> zadd w3cschool.cc 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 mongodb
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 rabitmq
(integer) 1
redis 127.0.0.1:6379> zadd w3cschool.cc 0 rabitmq
(integer) 0
redis 127.0.0.1:6379> ZRANGEBYSCORE w3cschool.cc 0 1000

1) "redis"
2) "mongodb"
3) "rabitmq"
```

Redis 命令

Redis 命令用于在 redis 服务上执行操作。

要在 redis 服务上执行命令需要一个 redis 客户端。Redis 客户端在我们之前下载的的 redis 的安装包中。

语法

Redis 客户端的基本语法为：

```
$ redis-cli
```

实例

以下实例讲解了如何启动 redis 客户端：

启动 redis 客户端，打开终端并输入命令 **redis-cli**。该命令会连接本地的 redis 服务。

```
$redis-cli
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

在以上实例中我们连接到本地的 redis 服务并执行 **PING** 命令，该命令用于检测 redis 服务是否启动。

在远程服务上执行命令

如果需要在远程 redis 服务上执行命令，同样我们使用的也是 **redis-cli** 命令。

语法

```
$ redis-cli -h host -p port -a password
```

实例

以下实例演示了如何连接到主机为 127.0.0.1，端口为 6379 ，密码为 mypass 的 redis 服务上。

```
$redis-cli -h 127.0.0.1 -p 6379 -a "mypass"
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

Redis 数据备份与恢复

Redis **SAVE** 命令用于创建当前数据库的备份。

语法

redis Save 命令基本语法如下：

```
redis 127.0.0.1:6379> SAVE
```

实例

```
redis 127.0.0.1:6379> SAVE
OK
```

该命令将在 redis 安装目录中创建dump.rdb文件。

恢复数据

如果需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可。获取 redis 目录可以使用 **CONFIG** 命令，如下所示：

```
redis 127.0.0.1:6379> CONFIG GET dir
1) "dir"
2) "/usr/local/redis/bin"
```

以上命令 **CONFIG GET dir** 输出的 redis 安装目录为 /usr/local/redis/bin。

Bgsave

创建 redis 备份文件也可以使用命令 **BGSAVE**，该命令在后台执行。

实例

```
127.0.0.1:6379> BGSAVE

Background saving started
```

Redis 安全

我们可以通过 redis 的配置文件设置密码参数，这样客户端连接到 redis 服务就需要密码验证，这样可以让你的 redis 服务更安全。

实例

我们可以通过以下命令查看是否设置了密码验证：

```
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) ""
```

默认情况下 requirepass 参数是空的，这就意味着你无需通过密码验证就可以连接到 redis 服务。

你可以通过以下命令来修改该参数：

```
127.0.0.1:6379> CONFIG set requirepass "w3cschool.cc"
OK
127.0.0.1:6379> CONFIG get requirepass
1) "requirepass"
2) "w3cschool.cc"
```

设置密码后，客户端连接 redis 服务就需要密码验证，否则无法执行命令。

语法

AUTH 命令基本语法格式如下：

```
127.0.0.1:6379> AUTH password
```

实例

```
127.0.0.1:6379> AUTH "w3cschool.cc"
OK
127.0.0.1:6379> SET mykey "Test value"
OK
127.0.0.1:6379> GET mykey
"Test value"
```

Redis 性能测试

Redis 性能测试是通过同时执行多个命令实现的。

语法

redis 性能测试的基本命令如下：

```
redis-benchmark [option] [option value]
```

实例

以下实例同时执行 10000 个请求来检测性能：

```
redis-benchmark -n 100000

PING_INLINE: 141043.72 requests per second
PING_BULK: 142857.14 requests per second
SET: 141442.72 requests per second
GET: 145348.83 requests per second
INCR: 137362.64 requests per second
LPUSH: 145348.83 requests per second
LPOP: 146198.83 requests per second
SADD: 146198.83 requests per second
SPOP: 149253.73 requests per second
LPUSH (needed to benchmark LRANGE): 148588.42 requests per second
LRANGE_100 (first 100 elements): 58411.21 requests per second
LRANGE_300 (first 300 elements): 21195.42 requests per second
LRANGE_500 (first 450 elements): 14539.11 requests per second
LRANGE_600 (first 600 elements): 10504.20 requests per second
MSET (10 keys): 93283.58 requests per second
```

redis 性能测试工具可选参数如下所示：

序号	选项	描述	默认值
1	**-h**	指定服务器主机名	127.0.0.1
2	**-p**	指定服务器端口	6379
3	**-s**	指定服务器 socket	
4	**-c**	指定并发连接数	50
5	**-n**	指定请求数	10000
6	**-d**	以字节的形式指定 SET/GET 值的数据大小	2
7	**-k**	1=keep alive 0=reconnect	1
8	**-r**	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	**-P**	通过管道传输 `<numreq>` 请求	1
10	**-q**	强制退出 redis。仅显示 query/sec 值	
11	**--csv**	以 CSV 格式输出	
12	**-l**	生成循环，永久执行测试	
13	**-t**	仅运行以逗号分隔的测试命令列表。	
14	**-I**	Idle 模式。仅打开 N 个 idle 连接并等待。	

实例

以下实例我们使用了多个参数来测试 redis 性能：

```
redis-benchmark -h 127.0.0.1 -p 6379 -t set,lpush -n 100000 -q

SET: 146198.83 requests per second
LPUSH: 145560.41 requests per second
```

以上实例中主机为 127.0.0.1，端口号为 6379，执行的命令为 set,lpush，请求数为 10000，通过 -q 参数让结果只显示每秒执行的请求数。

Redis 客户端连接

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

- 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。
- 然后为这个 socket 设置 TCP_NODELAY 属性，禁用 Nagle 算法
- 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

最大连接数

在 Redis2.4 中，最大连接数是被直接硬编码在代码里面的，而在2.6版本中这个值变成可配置的。

maxclients 的默认值是 10000，你也可以在 redis.conf 中对这个值进行修改。

```
config get maxclients

1) "maxclients"
2) "10000"
```

实例

以下实例我们在服务启动时设置最大连接数为 100000：

```
redis-server --maxclients 100000
```

客户端命令

命令	描述
CLIENT LIST	返回连接到 redis 服务的客户端列表
CLIENT SETNAME	设置当前连接的名称
CLIENT GETNAME	获取通过 CLIENT SETNAME 命令设置的服务名称
CLIENT PAUSE	挂起客户端连接，指定挂起的时间以毫秒计
CLIENT KILL	关闭客户端连接

Redis 管道技术

Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。这意味着通常情况下一个请求会遵循以下步骤：

- 客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应。
- 服务端处理命令，并将结果返回给客户端。

Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

实例

查看 redis 管道，只需要启动 redis 实例并输入以下命令：

```
$(echo -en "PING\r\n SET w3ckey redis\r\nGET w3ckey\r\nINCR visitor

+PONG
+OK
redis
:1
:2
:3
```

以上实例中我们通过使用 **PING** 命令查看redis服务是否可用， 之后我们们设置了 w3ckey 的值为 redis，然后我们获取 w3ckey 的值并使得 visitor 自增 3 次。

在返回的结果中我们可以看到这些命令一次性向 redis 服务提交，并最终一次性读取所有服务端的响应

管道技术的优势

管道技术最显著的优势是提高了 redis 服务的性能。

一些测试数据

在下面的测试中，我们将使用Redis的Ruby客户端，支持管道技术特性，测试管道技术对速度的提升效果。

```
require 'rubygems'
require 'redis'
def bench(descr)
start = Time.now
yield
puts "#{descr} #{Time.now-start} seconds"
end
def without_pipelining
r = Redis.new
10000.times {
    r.ping
}
end
def with_pipelining
r = Redis.new
r.pipelined {
    10000.times {
        r.ping
    }
}
end
bench("without pipelining") {
    without_pipelining
}
bench("with pipelining") {
    with_pipelining
}
```

从处于局域网中的Mac OS X系统上执行上面这个简单脚本的数据表明，开启了管道操作后，往返时延已经被改善得相当低了。

```
without pipelining 1.185238 seconds
with pipelining 0.250783 seconds
```

如你所见，开启管道后，我们的速度效率提升了5倍。

Redis 分区

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。

分区的优势

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

分区的不足

redis的一些特性在分区方面表现的不是很好：

- 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
- 涉及多个key的redis事务不能使用。
- 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
- 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

分区类型

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

范围分区

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。

比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。

这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

哈希分区

另外一种分区方法是hash分区。这对任何key都适用，也无需是object_name:<id>这种形式，像下面描述的一样简单：</id>

- 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
- 对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

Java 使用 Redis

安装

开始在 Java 中使用 Redis 前， 我们需要确保已经安装了 redis 服务及 Java redis 驱动，且你的机器上能正常使用 Java。 Java的安装配置可以参考我们的 Java开发环境配置 接下来让我们安装 Java redis 驱动：

- 首先你需要下载驱动包，下载 **jedis.jar**，确保下载最新驱动包。
- 在你的classpath中包含该驱动包。

连接到 redis 服务

```
import redis.clients.jedis.Jedis;
public class RedisJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //查看服务是否运行
      System.out.println("Server is running: "+jedis.ping());
 }
}
```

编译以上 Java 程序，确保驱动包的路径是正确的。

```
$javac RedisJava.java
$java RedisJava
Connection to server sucessfully
Server is running: PONG

Redis Java String Example
```

Redis Java String(字符串) 实例

```
import redis.clients.jedis.Jedis;
public class RedisStringJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //设置 redis 字符串数据
      jedis.set("w3ckey", "Redis tutorial");
     // 获取存储的数据并输出
     System.out.println("Stored string in redis:: "+ jedis.get("w3c
 }
}
```

编译以上程序。

```
$javac RedisStringJava.java
$java RedisStringJava
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List(列表) 实例

```
import redis.clients.jedis.Jedis;
public class RedisListJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //存储数据到列表中
      jedis.lpush("tutorial-list", "Redis");
      jedis.lpush("tutorial-list", "Mongodb");
      jedis.lpush("tutorial-list", "Mysql");
     // 获取存储的数据并输出
     List<String> list = jedis.lrange("tutorial-list", 0 ,5);
     for(int i=0; i<list.size(); i++) {
       System.out.println("Stored string in redis:: "+list.get(i));
     }
 }
}
```

编译以上程序。

```
$javac RedisListJava.java
$java RedisListJava
Connection to server sucessfully
Stored string in redis:: Redis
Stored string in redis:: Mongodb
Stored string in redis:: Mysql
```

Redis Java Keys 实例

```
import redis.clients.jedis.Jedis;
public class RedisKeyJava {
   public static void main(String[] args) {
      //连接本地的 Redis 服务
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");

     // 获取数据并输出
     List<String> list = jedis.keys("*");
     for(int i=0; i<list.size(); i++) {
       System.out.println("List of stored keys:: "+list.get(i));
     }
   }
}
```

编译以上程序。

```
$javac RedisKeyJava.java
$java RedisKeyJava
Connection to server sucessfully
List of stored keys:: tutorial-name
List of stored keys:: tutorial-list
```

PHP 使用 Redis

安装

开始在 PHP 中使用 Redis 前， 我们需要确保已经安装了 redis 服务及 PHP redis 驱动，且你的机器上能正常使用 PHP。 接下来让我们安装 PHP redis 驱动：下载地址为:**https://github.com/nicolasff/phpredis**。

PHP安装redis扩展

```
/usr/local/php/bin/phpize              #php安装后的路径

./configure --with-php-config=/usr/local/php/bin/php-config

make && make install
```

修改php.ini文件

```
vi /usr/local/php/lib/php.ini
```

增加如下内容:

```
extension_dir = "/usr/local/php/lib/php/extensions/no-debug-zts-20(

extension=redis.so
```

安装完成后重启php-fpm 或 apache。查看phpinfo信息，就能看到redis扩展。

Phar based on pear/PHP_Archive, original concept by Davey Shafik.
Phar fully realized by Gregory Beaver and Marcus Boerger.
Portions of tar implementation Copyright (c) 2003-2009 Tim Kientzle.

redis 第 1 条，共 6 条

Directive	Local Value	Master Value
phar.cache_list	*no value*	*no value*
phar.readonly	On	On
phar.require_hash	On	On

posix

Revision	$Id: 1dfa9997ed76804e53c91e0ce862f3707617b6ed $

redis

Redis Support	enabled
Redis Version	2.2.4

连接到 redis 服务

```
<?php
    //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
        //查看服务是否运行
   echo "Server is running: "+ $redis->ping();
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Server is running: PONG
```

Redis Java String(字符串) 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //设置 redis 字符串数据
   $redis->set("tutorial-name", "Redis tutorial");
   // 获取存储的数据并输出
   echo "Stored string in redis:: " + jedis.get("tutorial-name");
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List(列表) 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //存储数据到列表中
   $redis->lpush("tutorial-list", "Redis");
   $redis->lpush("tutorial-list", "Mongodb");
   $redis->lpush("tutorial-list", "Mysql");
   // 获取存储的数据并输出
   $arList = $redis->lrange("tutorial-list", 0 ,5);
   echo "Stored string in redis:: "
   print_r($arList);
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis::
Redis
Mongodb
Mysql
```

Redis Java Keys 实例

```
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   // 获取数据并输出
   $arList = $redis->keys("*");
   echo "Stored keys in redis:: "
   print_r($arList);
?>
```

执行脚本，输出结果为：

```
Connection to server sucessfully
Stored string in redis::
tutorial-name
tutorial-list
```

DEL

DEL key [key ...]

删除给定的一个或多个 `key` 。

不存在的 `key` 会被忽略。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为被删除的 `key` 的数量。删除单个字符串类型的 `key` ，时间复杂度为O(1)。删除单个列表、集合、有序集合或哈希表类型的 `key` ，时间复杂度为O(M)， `M` 为以上数据结构内的元素数量。

返回值：

被删除 `key` 的数量。

```
#  删除单个 key

redis> SET name huangz
OK

redis> DEL name
(integer) 1

# 删除一个不存在的 key

redis> EXISTS phone
(integer) 0

redis> DEL phone # 失败，没有 key 被删除
(integer) 0

# 同时删除多个 key

redis> SET name "redis"
OK

redis> SET type "key-value store"
OK

redis> SET website "redis.com"
OK

redis> DEL name type website
(integer) 3
```

DUMP

DUMP key

序列化给定 `key` ，并返回被序列化的值，使用 *RESTORE* 命令可以将这个值反序列化为 Redis 键。

序列化生成的值有以下几个特点：

- 它带有 64 位的校验和，用于检测错误， *RESTORE* 在进行反序列化之前会先检查校验和。
- 值的编码格式和 RDB 文件保持一致。
- RDB 版本会被编码在序列化值当中，如果因为 Redis 的版本不同造成 RDB 格式不兼容，那么 Redis 会拒绝对这个值进行反序列化操作。

序列化的值不包括任何生存时间信息。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行序列化的复杂度为 O(N*M) ，其中 N 是构成 `key` 的 Redis 对象的数量，而 M 则是这些对象的平均大小。如果序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果 `key` 不存在，那么返回 `nil` 。否则，返回序列化之后的值。

```
redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

redis> DUMP not-exists-key
(nil)
```

Redis 命令参考

Key（键）

EXISTS

EXISTS key

检查给定 `key` 是否存在。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

若 `key` 存在，返回 `1` ，否则返回 `0` 。

```
redis> SET db "redis"
OK

redis> EXISTS db
(integer) 1

redis> DEL db
(integer) 1

redis> EXISTS db
(integer) 0
```

EXPIRE

EXPIRE key seconds

为给定 `key` 设置生存时间，当 `key` 过期时(生存时间为 `0`)，它会被自动删除。

在 Redis 中，带有生存时间的 `key` 被称为『易失的』(volatile)。

生存时间可以通过使用 *DEL* 命令来删除整个 `key` 来移除，或者被 *SET* 和 *GETSET* 命令覆写(overwrite)，这意味着，如果一个命令只是修改(alter)一个带生存时间的 `key` 的值而不是用一个新的 `key` 值来代替(replace)它的话，那么生存时间不会被改变。

比如说，对一个 `key` 执行 *INCR* 命令，对一个列表进行 *LPUSH* 命令，或者对一个哈希表执行 *HSET* 命令，这类操作都不会修改 `key` 本身的生存时间。

另一方面，如果使用 *RENAME* 对一个 `key` 进行改名，那么改名后的 `key` 的生存时间和改名前一样。

RENAME 命令的另一种可能是，尝试将一个带生存时间的 `key` 改名成另一个带生存时间的 `another_key` ，这时旧的 `another_key` (以及它的生存时间)会被删除，然后旧的 `key` 会改名为 `another_key` ，因此，新的 `another_key` 的生存时间也和原本的 `key` 一样。

使用 *PERSIST* 命令可以在不删除 `key` 的情况下，移除 `key` 的生存时间，让 `key` 重新成为一个『持久的』(persistent) `key` 。

更新生存时间

可以对一个已经带有生存时间的 `key` 执行 *EXPIRE* 命令，新指定的生存时间会取代旧的生存时间。

过期时间的精确度

在 Redis 2.4 版本中，过期时间的延迟在 1 秒钟之内 —— 也即是，就算 `key` 已经过期，但它还是可能在过期之后一秒钟之内被访问到，而在新的 Redis 2.6 版本中，延迟被降低到 1 毫秒之内。

Redis 2.1.3 之前的不同之处

在 Redis 2.1.3 之前的版本中，修改一个带有生存时间的 `key` 会导致整个 `key` 被删除，这一行为是受当时复制(replication)层的限制而作出的，现在这一限制已经被修复。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功返回 `1` 。当 `key` 不存在或者不能为 `key` 设置生存时间时(比如在低于 2.1.3 版本的 Redis 中你尝试更新 `key` 的生存时间)，返回 `0` 。

```
redis> SET cache_page "www.google.com"
OK

redis> EXPIRE cache_page 30  # 设置过期时间为 30 秒
(integer) 1

redis> TTL cache_page    # 查看剩余生存时间
(integer) 23

redis> EXPIRE cache_page 30000   # 更新过期时间
(integer) 1

redis> TTL cache_page
(integer) 29996
```

模式：导航会话

假设你有一项 web 服务，打算根据用户最近访问的 N 个页面来进行物品推荐，并且假设用户停止阅览超过 60 秒，那么就清空阅览记录(为了减少物品推荐的计算量，并且保持推荐物品的新鲜度)。

这些最近访问的页面记录，我们称之为『导航会话』(Navigation session)，可以用 *INCR* 和 *RPUSH* 命令在 Redis 中实现它：每当用户阅览一个网页的时候，执行以下代码：

```
MULTI
    RPUSH pagewviews.user:<userid> http://.....
    EXPIRE pagewviews.user:<userid> 60
EXEC
```

如果用户停止阅览超过 60 秒，那么它的导航会话就会被清空，当用户重新开始阅览的时候，系统又会重新记录导航会话，继续进行物品推荐。

EXPIREAT

EXPIREAT key timestamp

EXPIREAT 的作用和 *EXPIRE* 类似，都用于为 `key` 设置生存时间。

不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间，返回 `0` 。

```
redis> SET cache www.google.com
OK

redis> EXPIREAT cache 1355292000     # 这个 key 将在 2012.12.12 过期

(integer) 1

redis> TTL cache
(integer) 45081860
```

KEYS

KEYS pattern

查找所有符合给定模式 `pattern` 的 `key` 。

`KEYS *` 匹配数据库中所有 `key` 。 `KEYS h?llo` 匹配 `hello` ， `hallo` 和 `hxllo` 等。 `KEYS h*llo` 匹配 `hllo` 和 `heeeeello` 等。 `KEYS h[ae]llo` 匹配 `hello` 和 `hallo` ，但不匹配 `hillo` 。

特殊符号用 `\` 隔开

Warning

KEYS 的速度非常快，但在一个大的数据库中使用它仍然可能造成性能问题，如果你需要从一个数据集中查找特定的 `key` ，你最好还是用 Redis 的集合结构(set)来代替。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为数据库中 `key` 的数量。

返回值：

符合给定模式的 `key` 列表。

```
redis> MSET one 1 two 2 three 3 four 4  # 一次设置 4 个 key
OK

redis> KEYS *o*
1) "four"
2) "two"
3) "one"

redis> KEYS t??
1) "two"

redis> KEYS t[w]*
1) "two"

redis> KEYS *  # 匹配数据库内所有 key
1) "four"
2) "three"
3) "two"
4) "one"
```

MIGRATE

MIGRATE host port key destination-db timeout [COPY] [REPLACE]

将 `key` 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功，`key` 保证会出现在目标实例上，而当前实例上的 `key` 会被删除。

这个命令是一个原子操作，它在执行的时候会阻塞进行迁移的两个实例，直到以下任意结果发生：迁移成功，迁移失败，等到超时。

命令的内部实现是这样的：它在当前实例对给定 `key` 执行 *DUMP* 命令，将它序列化，然后传送到目标实例，目标实例再使用 *RESTORE* 对数据进行反序列化，并将反序列化所得的数据添加到数据库中；当前实例就像目标实例的客户端那样，只要看到 *RESTORE* 命令返回 `OK` ，它就会调用 *DEL* 删除自己数据库上的 `key` 。

`timeout` 参数以毫秒为格式，指定当前实例和目标实例进行沟通的最大间隔时间。这说明操作并不一定要在 `timeout` 毫秒内完成，只是说数据传送的时间不能超过这个 `timeout` 数。

MIGRATE 命令需要在给定的时间规定内完成 IO 操作。如果在传送数据时发生 IO 错误，或者达到了超时时间，那么命令会停止执行，并返回一个特殊的错误：`IOERR` 。

当 `IOERR` 出现时，有以下两种可能：

- `key` 可能存在于两个实例
- `key` 可能只存在于当前实例

唯一不可能发生的情况就是丢失 `key` ，因此，如果一个客户端执行 MIGRATE 命令，并且不幸遇上 `IOERR` 错误，那么这个客户端唯一要做的就是检查自己数据库上的 `key` 是否已经被正确地删除。

如果有其他错误发生，那么 MIGRATE 保证 `key` 只会出现在当前实例中。（当然，目标实例的给定数据库上可能有和 `key` 同名的键，不过这和 MIGRATE 命令没有关系）。

可选项：

- `COPY` ：不移除源实例上的 `key` 。
- `REPLACE` ：替换目标实例上已存在的 `key` 。

可用版本：

>= 2.6.0

时间复杂度：

这个命令在源实例上实际执行 *DUMP* 命令和 *DEL* 命令，在目标实例执行 *RESTORE* 命令，查看以上命令的文档可以看到详细的复杂度说明。 `key` 数据在两个实例之间传输的复杂度为 O(N) 。

返回值：

迁移成功时返回 `OK` ，否则返回相应的错误。

示例

先启动两个 Redis 实例，一个使用默认的 6379 端口，一个使用 7777 端口。

```
$ ./redis-server &
[1] 3557

...

$ ./redis-server --port 7777 &
[2] 3560

...
```

然后用客户端连上 6379 端口的实例，设置一个键，然后将它迁移到 7777 端口的实例上：

```
$ ./redis-cli

redis 127.0.0.1:6379> flushdb
OK

redis 127.0.0.1:6379> SET greeting "Hello from 6379 instance"
OK

redis 127.0.0.1:6379> MIGRATE 127.0.0.1 7777 greeting 0 1000
OK

redis 127.0.0.1:6379> EXISTS greeting                              #
(integer) 0
```

使用另一个客户端，查看 7777 端口上的实例：

```
$ ./redis-cli -p 7777

redis 127.0.0.1:7777> GET greeting
"Hello from 6379 instance"
```

MOVE

MOVE key db

将当前数据库的 `key` 移动到给定的数据库 `db` 当中。

如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 `key` ，或者 `key` 不存在于当前数据库，那么 `MOVE` 没有任何效果。

因此，也可以利用这一特性，将 MOVE 当作锁(locking)原语(primitive)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

移动成功返回 `1` ，失败则返回 `0` 。

```
# key 存在于当前数据库

redis> SELECT 0                             # redis默认使用数据库 0,
OK

redis> SET song "secret base - Zone"
OK

redis> MOVE song 1                          # 将 song 移动到数据库 1
(integer) 1

redis> EXISTS song                          # song 已经被移走
(integer) 0

redis> SELECT 1                             # 使用数据库 1
OK

redis:1> EXISTS song                        # 证实 song 被移到了数据库
(integer) 1

# 当 key 不存在的时候

redis:1> EXISTS fake_key
(integer) 0

redis:1> MOVE fake_key 0                    # 试图从数据库 1 移动一个不
(integer) 0
```

```
redis:1> select 0                                   # 使用数据库0
OK

redis> EXISTS fake_key                              # 证实 fake_key 不存在
(integer) 0

# 当源数据库和目标数据库有相同的 key 时

redis> SELECT 0                                     # 使用数据库0
OK
redis> SET favorite_fruit "banana"
OK

redis> SELECT 1                                     # 使用数据库1
OK
redis:1> SET favorite_fruit "apple"
OK

redis:1> SELECT 0                                   # 使用数据库0，并试图将 fa
OK

redis> MOVE favorite_fruit 1                        # 因为两个数据库有相同的 k
(integer) 0

redis> GET favorite_fruit                           # 数据库 0 的 favorite_f
"banana"

redis> SELECT 1
OK

redis:1> GET favorite_fruit                         # 数据库 1 的 favorite_f
"apple"
```

OBJECT

OBJECT subcommand [arguments [arguments]]

OBJECT 命令允许从内部察看给定 `key` 的 Redis 对象。

它通常用在除错(debugging)或者了解为了节省空间而对 `key` 使用特殊编码的情况。当将Redis用作缓存程序时，你也可以通过 OBJECT 命令中的信息，决定 `key` 的驱逐策略(eviction policies)。

OBJECT 命令有多个子命令：

- `OBJECT REFCOUNT <key>` 返回给定 `key` 引用所储存的值的次数。此命令主要用于除错。
- `OBJECT ENCODING <key>` 返回给定 `key` 锁储存的值所使用的内部表示(representation)。
- `OBJECT IDLETIME <key>` 返回给定 `key` 自储存以来的空闲时间(idle， 没有被读取也没有被写入)，以秒为单位。对象可以以多种方式编码：
- 字符串可以被编码为 `raw` (一般字符串)或 `int` (为了节约内存，Redis 会将字符串表示的 64 位有符号整数编码为整数来进行储存）。
- 列表可以被编码为 `ziplist` 或 `linkedlist` 。 `ziplist` 是为节约大小较小的列表空间而作的特殊表示。
- 集合可以被编码为 `intset` 或者 `hashtable` 。 `intset` 是只储存数字的小集合的特殊表示。
- 哈希表可以编码为 `zipmap` 或者 `hashtable` 。 `zipmap` 是小哈希表的特殊表示。
- 有序集合可以被编码为 `ziplist` 或者 `skiplist` 格式。 `ziplist` 用于表示小的有序集合，而 `skiplist` 则用于表示任何大小的有序集合。假如你做了什么让 Redis 没办法再使用节省空间的编码时(比如将一个只有 1 个元素的集合扩展为一个有 100 万个元素的集合)，特殊编码类型(specially encoded types)会自动转换成通用类型(general type)。

可用版本：

>= 2.2.3

时间复杂度：

O(1)

返回值：

`REFCOUNT` 和 `IDLETIME` 返回数字。 `ENCODING` 返回相应的编码类型。

```
redis> SET game "COD"           # 设置一个字符串
OK

redis> OBJECT REFCOUNT game     # 只有一个引用
(integer) 1

redis> OBJECT IDLETIME game     # 等待一阵。。。然后查看空闲时间
(integer) 90

redis> GET game                 # 提取game， 让它处于活跃(active)状态
"COD"

redis> OBJECT IDLETIME game     # 不再处于空闲状态
(integer) 0

redis> OBJECT ENCODING game     # 字符串的编码方式
"raw"

redis> SET big-number 23102930128301091820391092019203810281029831(
OK

redis> OBJECT ENCODING big-number
"raw"

redis> SET small-number 12345  # 而短的数字则会被编码为整数
OK

redis> OBJECT ENCODING small-number
"int"
```

PERSIST

PERSIST key

移除给定 `key` 的生存时间，将这个 `key` 从『易失的』(带生存时间 `key`)转换成『持久的』(一个不带生存时间、永不过期的 `key`)。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

当生存时间移除成功时，返回 `1` .如果 `key` 不存在或 `key` 没有设置生存时间，返回 `0` 。

```
redis> SET mykey "Hello"
OK

redis> EXPIRE mykey 10  # 为 key 设置生存时间
(integer) 1

redis> TTL mykey
(integer) 10

redis> PERSIST mykey    # 移除 key 的生存时间
(integer) 1

redis> TTL mykey
(integer) -1
```

PEXPIRE

PEXPIRE key milliseconds

这个命令和 *EXPIRE* 命令的作用类似，但是它以毫秒为单位设置 `key` 的生存时间，而不像 *EXPIRE* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1``key` 不存在或设置失败，返回 `0`

```
redis> SET mykey "Hello"
OK

redis> PEXPIRE mykey 1500
(integer) 1

redis> TTL mykey    # TTL 的返回值以秒为单位
(integer) 2

redis> PTTL mykey   # PTTL 可以给出准确的毫秒数
(integer) 1499
```

PEXPIREAT

PEXPIREAT key milliseconds-timestamp

这个命令和 *EXPIREAT* 命令类似，但它以毫秒为单位设置 `key` 的过期 unix 时间戳，而不是像 *EXPIREAT* 那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间时，返回 `0` 。(查看 *EXPIRE* 命令获取更多信息)

```
redis> SET mykey "Hello"
OK

redis> PEXPIREAT mykey 1555555555005
(integer) 1

redis> TTL mykey           # TTL 返回秒
(integer) 223157079

redis> PTTL mykey          # PTTL 返回毫秒
(integer) 223157079318
```

PTTL

PTTL key

这个命令类似于 *TTL* 命令，但它以毫秒为单位返回 `key` 的剩余生存时间，而不是像 *TTL* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以毫秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> PTTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> PTTL key
(integer) -1

# 有剩余生存时间的 key

redis> PEXPIRE key 10086
(integer) 1

redis> PTTL key
(integer) 6179
```

RANDOMKEY

RANDOMKEY

从当前数据库中随机返回(不删除)一个 `key` 。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当数据库不为空时，返回一个 `key` 。当数据库为空时，返回 `nil` 。

```
# 数据库不为空

redis> MSET fruit "apple" drink "beer" food "cookies"   # 设置多个 k
OK

redis> RANDOMKEY
"fruit"

redis> RANDOMKEY
"food"

redis> KEYS *    # 查看数据库内所有key，证明 RANDOMKEY 并不删除 key
1) "food"
2) "drink"
3) "fruit"

# 数据库为空

redis> FLUSHDB  # 删除当前数据库所有 key
OK

redis> RANDOMKEY
(nil)
```

RENAME

RENAME key newkey

将 `key` 改名为 `newkey` 。

当 `key` 和 `newkey` 相同，或者 `key` 不存在时，返回一个错误。

当 `newkey` 已经存在时， RENAME 命令将覆盖旧值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

改名成功时提示 `OK` ，失败时候返回一个错误。

```
# key 存在且 newkey 不存在

redis> SET message "hello world"
OK

redis> RENAME message greeting
OK

redis> EXISTS message               # message 不复存在
(integer) 0

redis> EXISTS greeting              # greeting 取而代之
(integer) 1

# 当 key 不存在时，返回错误

redis> RENAME fake_key never_exists
(error) ERR no such key

# newkey 已存在时， RENAME 会覆盖旧 newkey

redis> SET pc "lenovo"
OK

redis> SET personal_computer "dell"
OK

redis> RENAME pc personal_computer
OK

redis> GET pc
(nil)

redis:1> GET personal_computer      # 原来的值 dell 被覆盖了
"lenovo"
```

RENAMENX

RENAMENX key newkey

当且仅当 `newkey` 不存在时，将 `key` 改名为 `newkey` 。

当 `key` 不存在时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

修改成功时，返回 `1` 。如果 `newkey` 已经存在，返回 `0` 。

```
# newkey 不存在，改名成功

redis> SET player "MPlyaer"
OK

redis> EXISTS best_player
(integer) 0

redis> RENAMENX player best_player
(integer) 1

# newkey存在时，失败

redis> SET animal "bear"
OK

redis> SET favorite_animal "butterfly"
OK

redis> RENAMENX animal favorite_animal
(integer) 0

redis> get animal
"bear"

redis> get favorite_animal
"butterfly"
```

RESTORE

RESTORE key ttl serialized-value [REPLACE]

反序列化给定的序列化值，并将它和给定的 `key` 关联。

参数 `ttl` 以毫秒为单位为 `key` 设置生存时间；如果 `ttl` 为 `0` ，那么不设置生存时间。

RESTORE 在执行反序列化之前会先对序列化值的 RDB 版本和数据校验和进行检查，如果 RDB 版本不相同或者数据不完整的话，那么 RESTORE 会拒绝进行反序列化，并返回一个错误。

如果键 `key` 已经存在， 并且给定了 `REPLACE` 选项， 那么使用反序列化得出的值来代替键 `key` 原有的值； 相反地， 如果键 `key` 已经存在， 但是没有给定 `REPLACE` 选项， 那么命令返回一个错误。

更多信息可以参考 *DUMP* 命令。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行反序列化的复杂度为 O(N*M*) ，其中 *N* 是构成 `key` 的 *Redis* 对象的数量，而 *M* 则是这些对象的平均大小。有序集合*(sorted set)*的反序列化复杂度为 *O(N*M*log(N))* ，因为有序集合每次插入的复杂度为 O(log(N)) 。如果反序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果反序列化成功那么返回 `OK` ，否则返回一个错误。

```
# 创建一个键，作为 DUMP 命令的输入

redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

# 将序列化数据 RESTORE 到另一个键上面

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
OK

redis> GET greeting-again
"hello, dumping world!"

# 在没有给定 REPLACE 选项的情况下，再次尝试反序列化到同一个键，失败

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
(error) ERR Target key name is busy.

# 给定 REPLACE 选项，对同一个键进行反序列化成功

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\
OK

# 尝试使用无效的值进行反序列化，出错

redis> RESTORE fake-message 0 "hello moto moto blah blah"
(error) ERR DUMP payload version or checksum are wrong
```

SORT

SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC | DESC] [ALPHA] [STORE destination]

返回或保存给定列表、集合、有序集合 `key` 中经过排序的元素。

排序默认以数字作为对象，值被解释为双精度浮点数，然后进行比较。

一般 SORT 用法

最简单的 SORT 使用方法是 `SORT key` 和 `SORT key DESC` ：

- `SORT key` 返回键值从小到大排序的结果。
- `SORT key DESC` 返回键值从大到小排序的结果。

假设 `today_cost` 列表保存了今日的开销金额， 那么可以用 SORT 命令对它进行排序：

```
# 开销金额列表

redis> LPUSH today_cost 30 1.5 10 8
(integer) 4

# 排序

redis> SORT today_cost
1) "1.5"
2) "8"
3) "10"
4) "30"

# 逆序排序

redis 127.0.0.1:6379> SORT today_cost DESC
1) "30"
2) "10"
3) "8"
4) "1.5"
```

使用 ALPHA 修饰符对字符串进行排序

因为 SORT 命令默认排序对象为数字， 当需要对字符串进行排序时， 需要显式地在 SORT 命令之后添加 `ALPHA` 修饰符：

```
# 网址

redis> LPUSH website "www.reddit.com"
(integer) 1

redis> LPUSH website "www.slashdot.com"
(integer) 2

redis> LPUSH website "www.infoq.com"
(integer) 3

# 默认（按数字）排序

redis> SORT website
1) "www.infoq.com"
2) "www.slashdot.com"
3) "www.reddit.com"

# 按字符排序

redis> SORT website ALPHA
1) "www.infoq.com"
2) "www.reddit.com"
3) "www.slashdot.com"
```

如果系统正确地设置了 `LC_COLLATE` 环境变量的话，Redis能识别 `UTF-8` 编码。

使用 LIMIT 修饰符限制返回结果

排序之后返回元素的数量可以通过 `LIMIT` 修饰符进行限制， 修饰符接受 `offset` 和 `count` 两个参数：

- `offset` 指定要跳过的元素数量。
- `count` 指定跳过 `offset` 个指定的元素之后，要返回多少个对象。

以下例子返回排序结果的前 5 个对象(`offset` 为 `0` 表示没有元素被跳过)。

```
# 添加测试数据，列表值为 1 指 10

redis 127.0.0.1:6379> RPUSH rank 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH rank 2 4 6 8 10
(integer) 10

# 返回列表中最小的 5 个值

redis 127.0.0.1:6379> SORT rank LIMIT 0 5
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

可以组合使用多个修饰符。以下例子返回从大到小排序的前 5 个对象。

```
redis 127.0.0.1:6379> SORT rank LIMIT 0 5 DESC
1) "10"
2) "9"
3) "8"
4) "7"
5) "6"
```

使用外部 key 进行排序

可以使用外部 `key` 的数据作为权重，代替默认的直接对比键值的方式来进行排序。

假设现在有用户数据如下：

uid	user*name*{uid}	user*level*{uid}
1	admin	9999
2	jack	10
3	peter	25
4	mary	70

以下代码将数据输入到 Redis 中：

```
# admin

redis 127.0.0.1:6379> LPUSH uid 1
(integer) 1

redis 127.0.0.1:6379> SET user_name_1 admin
OK

redis 127.0.0.1:6379> SET user_level_1 9999
OK

# jack

redis 127.0.0.1:6379> LPUSH uid 2
(integer) 2

redis 127.0.0.1:6379> SET user_name_2 jack
OK

redis 127.0.0.1:6379> SET user_level_2 10
OK

# peter

redis 127.0.0.1:6379> LPUSH uid 3
(integer) 3

redis 127.0.0.1:6379> SET user_name_3 peter
OK

redis 127.0.0.1:6379> SET user_level_3 25
OK

# mary

redis 127.0.0.1:6379> LPUSH uid 4
(integer) 4

redis 127.0.0.1:6379> SET user_name_4 mary
OK

redis 127.0.0.1:6379> SET user_level_4 70
OK
```

BY 选项

默认情况下， `SORT uid` 直接按 `uid` 中的值排序：

```
redis 127.0.0.1:6379> SORT uid
1) "1"      # admin
2) "2"      # jack
3) "3"      # peter
4) "4"      # mary
```

通过使用 `BY` 选项，可以让 `uid` 按其他键的元素来排序。

比如说， 以下代码让 `uid` 键按照 `user_level_{uid}` 的大小来排序：

```
redis 127.0.0.1:6379> SORT uid BY user_level_*
1) "2"      # jack , level = 10
2) "3"      # peter, level = 25
3) "4"      # mary, level = 70
4) "1"      # admin, level = 9999
```

`user_level_*` 是一个占位符， 它先取出 `uid` 中的值， 然后再用这个值来查找相应的键。

比如在对 `uid` 列表进行排序时， 程序就会先取出 `uid` 的值 `1` 、 `2` 、 `3` 、 `4` ， 然后使用 `user_level_1` 、 `user_level_2` 、 `user_level_3` 和 `user_level_4` 的值作为排序 `uid` 的权重。

GET 选项

使用 `GET` 选项， 可以根据排序的结果来取出相应的键值。

比如说， 以下代码先排序 `uid` ， 再取出键 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid GET user_name_*
1) "admin"
2) "jack"
3) "peter"
4) "mary"
```

组合使用 BY 和 GET

通过组合使用 `BY` 和 `GET` ， 可以让排序结果以更直观的方式显示出来。

比如说， 以下代码先按 `user_level_{uid}` 来排序 `uid` 列表， 再取出相应的 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid BY user_level_* GET user_name_*
1) "jack"       # level = 10
2) "peter"      # level = 25
3) "mary"       # level = 70
4) "admin"      # level = 9999
```

现在的排序结果要比只使用 `SORT uid BY user_level_*` 要直观得多。

获取多个外部键

可以同时使用多个 `GET` 选项， 获取多个外部键的值。

以下代码就按 `uid` 分别获取 `user_level_{uid}` 和 `user_name_{uid}` ：

```
redis 127.0.0.1:6379> SORT uid GET user_level_* GET user_name_*
1) "9999"       # level
2) "admin"      # name
3) "10"
4) "jack"
5) "25"
6) "peter"
7) "70"
8) "mary"
```

`GET` 有一个额外的参数规则，那就是 —— 可以用 `#` 获取被排序键的值。

以下代码就将 `uid` 的值、及其相应的 `user_level_*` 和 `user_name_*` 都返回为结果：

```
redis 127.0.0.1:6379> SORT uid GET # GET user_level_* GET user_name
1) "1"          # uid
2) "9999"       # level
3) "admin"      # name
4) "2"
5) "10"
6) "jack"
7) "3"
8) "25"
9) "peter"
10) "4"
11) "70"
12) "mary"
```

获取外部键，但不进行排序

通过将一个不存在的键作为参数传给 `BY` 选项， 可以让 `SORT` 跳过排序操作， 直接返回结果：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key
1) "4"
2) "3"
3) "2"
4) "1"
```

这种用法在单独使用时，没什么实际用处。

不过，通过将这种用法和 `GET` 选项配合， 就可以在不排序的情况下， 获取多个外部键， 相当于执行一个整合的获取操作（类似于 SQL 数据库的 `join` 关键字）。

以下代码演示了，如何在不引起排序的情况下，使用 `SORT` 、 `BY` 和 `GET` 获取多个外部键：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key GET # GET user_lev
1) "4"      # id
2) "70"     # level
3) "mary"   # name
4) "3"
5) "25"
6) "peter"
7) "2"
8) "10"
9) "jack"
10) "1"
11) "9999"
12) "admin"
```

将哈希表作为 GET 或 BY 的参数

除了可以将字符串键之外， 哈希表也可以作为 `GET` 或 `BY` 选项的参数来使用。

比如说，对于前面给出的用户信息表：

uid	user*name*{uid}	user*level*{uid}
1	admin	9999
2	jack	10
3	peter	25
4	mary	70

我们可以不将用户的名字和级别保存在 user_name_{uid} 和 user_level_{uid} 两个字符串键中， 而是用一个带有 name 域和 level 域的哈希表 user_info_{uid} 来保存用户的名字和级别信息：

```
redis 127.0.0.1:6379> HMSET user_info_1 name admin level 9999
OK

redis 127.0.0.1:6379> HMSET user_info_2 name jack level 10
OK

redis 127.0.0.1:6379> HMSET user_info_3 name peter level 25
OK

redis 127.0.0.1:6379> HMSET user_info_4 name mary level 70
OK
```

之后， BY 和 GET 选项都可以用 key->field 的格式来获取哈希表中的域的值， 其中 key 表示哈希表键， 而 field 则表示哈希表的域：

```
redis 127.0.0.1:6379> SORT uid BY user_info_*->level
1) "2"
2) "3"
3) "4"
4) "1"

redis 127.0.0.1:6379> SORT uid BY user_info_*->level GET user_info_
1) "jack"
2) "peter"
3) "mary"
4) "admin"
```

保存排序结果

默认情况下， SORT 操作只是简单地返回排序结果，并不进行任何保存操作。

通过给 STORE 选项指定一个 key 参数，可以将排序结果保存到给定的键上。

如果被指定的 key 已存在，那么原有的值将被排序结果覆盖。

```
# 测试数据

redis 127.0.0.1:6379> RPUSH numbers 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH numbers 2 4 6 8 10
(integer) 10

redis 127.0.0.1:6379> LRANGE numbers 0 -1
1) "1"
2) "3"
3) "5"
4) "7"
5) "9"
6) "2"
7) "4"
8) "6"
9) "8"
10) "10"

redis 127.0.0.1:6379> SORT numbers STORE sorted-numbers
(integer) 10

# 排序后的结果

redis 127.0.0.1:6379> LRANGE sorted-numbers 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
10) "10"
```

可以通过将 SORT 命令的执行结果保存，并用 *EXPIRE* 为结果设置生存时间，以此来产生一个 SORT 操作的结果缓存。

这样就可以避免对 SORT 操作的频繁调用：只有当结果集过期时，才需要再调用一次 SORT 操作。

另外，为了正确实现这一用法，你可能需要加锁以避免多个客户端同时进行缓存重建(也就是多个客户端，同一时间进行 SORT 操作，并保存为结果集)，具体参见 *SETNX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(N+M*log(M))， `N` 为要排序的列表或集合内的元素数量， `M` 为要返回的元素数量。如果只是使用 SORT 命令的 `GET` 选项获取数据而没有进行排序，时间复杂度 O(N)。

返回值：

没有使用 `STORE` 参数，返回列表形式的排序结果。使用 `STORE` 参数，返回排序结果的元素数量。

TTL

TTL key

以秒为单位，返回给定 `key` 的剩余生存时间(TTL, time to live)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> TTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> TTL key
(integer) -1

# 有剩余生存时间的 key

redis> EXPIRE key 10086
(integer) 1

redis> TTL key
(integer) 10084
```

TYPE

TYPE key

返回 `key` 所储存的值的类型。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`none` (key不存在) `string` (字符串) `list` (列表) `set` (集合) `zset` (有序集) `hash` (哈希表)

```
# 字符串

redis> SET weather "sunny"
OK

redis> TYPE weather
string

# 列表

redis> LPUSH book_list "programming in scala"
(integer) 1

redis> TYPE book_list
list

# 集合

redis> SADD pat "dog"
(integer) 1

redis> TYPE pat
set
```

SCAN

SCAN cursor [MATCH pattern] [COUNT count]

SCAN 命令及其相关的 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都用于增量地迭代（incrementally iterate）一集元素（a collection of elements）：

- *SCAN* 命令用于迭代当前数据库中的数据库键。
- *SSCAN* 命令用于迭代集合键中的元素。
- *HSCAN* 命令用于迭代哈希键中的键值对。
- *ZSCAN* 命令用于迭代有序集合中的元素（包括元素成员和元素分值）。

以上列出的四个命令都支持增量式迭代， 它们每次执行都只会返回少量元素， 所以这些命令可以用于生产环境， 而不会出现像 *KEYS* 命令、 *SMEMBERS* 命令带来的问题 —— 当 *KEYS* 命令被用于处理一个大的数据库时， 又或者 *SMEMBERS* 命令被用于处理一个大的集合键时， 它们可能会阻塞服务器达数秒之久。

不过， 增量式迭代命令也不是没有缺点的： 举个例子， 使用 *SMEMBERS* 命令可以返回集合键当前包含的所有元素， 但是对于 *SCAN* 这类增量式迭代命令来说， 因为在对键进行增量式迭代的过程中， 键可能会被修改， 所以增量式迭代命令只能对被返回的元素提供有限的保证 （offer limited guarantees about the returned elements）。

因为 *SCAN* 、 *SSCAN* 、 *HSCAN* 和 *ZSCAN* 四个命令的工作方式都非常相似， 所以这个文档会一并介绍这四个命令， 但是要记住：

- *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令的第一个参数总是一个数据库键。
- 而 *SCAN* 命令则不需要在第一个参数提供任何数据库键 —— 因为它迭代的是当前数据库中的所有数据库键。

SCAN 命令的基本用法

SCAN 命令是一个基于游标的迭代器（cursor based iterator）： *SCAN* 命令每次被调用之后， 都会向用户返回一个新的游标， 用户在下次迭代时需要使用这个新游标作为 *SCAN* 命令的游标参数， 以此来延续之前的迭代过程。

当 *SCAN* 命令的游标参数被设置为 `0` 时， 服务器将开始一次新的迭代， 而当服务器向用户返回值为 `0` 的游标时， 表示迭代已结束。

以下是一个 *SCAN* 命令的迭代过程示例：

```
redis 127.0.0.1:6379> scan 0
1) "17"
2)  1) "key:12"
    2) "key:8"
    3) "key:4"
    4) "key:14"
    5) "key:16"
    6) "key:17"
    7) "key:15"
    8) "key:10"
    9) "key:3"
    10) "key:7"
    11) "key:1"

redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18"
   3) "key:0"
   4) "key:2"
   5) "key:19"
   6) "key:13"
   7) "key:6"
   8) "key:9"
   9) "key:11"
```

在上面这个例子中，第一次迭代使用 `0` 作为游标，表示开始一次新的迭代。

第二次迭代使用的是第一次迭代时返回的游标，也即是命令回复第一个元素的值 —— `17` 。

从上面的示例可以看到，*SCAN* 命令的回复是一个包含两个元素的数组，第一个数组元素是用于进行下一次迭代的新游标，而第二个数组元素则是一个数组，这个数组中包含了所有被迭代的元素。

在第二次调用 *SCAN* 命令时，命令返回了游标 `0` ，这表示迭代已经结束，整个数据集（collection）已经被完整遍历过了。

以 `0` 作为游标开始一次新的迭代，一直调用 *SCAN* 命令，直到命令返回游标 `0` ，我们称这个过程为一次完整遍历（full iteration）。

SCAN 命令的保证（guarantees）

SCAN 命令，以及其他增量式迭代命令，在进行完整遍历的情况下可以为用户带来以下保证：从完整遍历开始直到完整遍历结束期间，一直存在于数据集内的所有元素都会被完整遍历返回；这意味着，如果有一个元素，它从遍历开始直到遍历结束期间都存在于被遍历的数据集当中，那么 *SCAN* 命令总会在某次迭代中将这个元素返回给用户。

然而因为增量式命令仅仅使用游标来记录迭代状态， 所以这些命令带有以下缺点：

- 同一个元素可能会被返回多次。 处理重复元素的工作交由应用程序负责， 比如说， 可以考虑将迭代返回的元素仅仅用于可以安全地重复执行多次的操作上。
- 如果一个元素是在迭代过程中被添加到数据集的， 又或者是在迭代过程中从数据集中被删除的， 那么这个元素可能会被返回， 也可能不会， 这是未定义的（undefined）。

SCAN 命令每次执行返回的元素数量

增量式迭代命令并不保证每次执行都返回某个给定数量的元素。

增量式命令甚至可能会返回零个元素， 但只要命令返回的游标不是 `0` ， 应用程序就不应该将迭代视作结束。

不过命令返回的元素数量总是符合一定规则的， 在实际中：

- 对于一个大数据集来说， 增量式迭代命令每次最多可能会返回数十个元素；
- 而对于一个足够小的数据集来说， 如果这个数据集的底层表示为编码数据结构（encoded data structure，适用于是小集合键、小哈希键和小有序集合键）， 那么增量迭代命令将在一次调用中返回数据集中的所有元素。

最后， 用户可以通过增量式迭代命令提供的 `COUNT` 选项来指定每次迭代返回元素的最大值。

COUNT 选项

虽然增量式迭代命令不保证每次迭代所返回的元素数量， 但我们可以使用 `COUNT` 选项， 对命令的行为进行一定程度上的调整。

基本上， `COUNT` 选项的作用就是让用户告知迭代命令， 在每次迭代中应该从数据集里返回多少元素。

虽然 `COUNT` 选项只是对增量式迭代命令的一种提示（hint）， 但是在大多数情况下， 这种提示都是有效的。

- `COUNT` 参数的默认值为 `10` 。
- 在迭代一个足够大的、由哈希表实现的数据库、集合键、哈希键或者有序集合键时， 如果用户没有使用 `MATCH` 选项， 那么命令返回的元素数量通常和 `COUNT` 选项指定的一样， 或者比 `COUNT` 选项指定的数量稍多一些。
- 在迭代一个编码为整数集合（intset，一个只由整数值构成的小集合）、 或者编码为压缩列表（ziplist，由不同值构成的一个小哈希或者一个小有序集合）时， 增量式迭代命令通常会无视 `COUNT` 选项指定的值， 在第一次迭代就将数据集包含的所有元素都返回给用户。

Note

并非每次迭代都要使用相同的 `COUNT` 值。

用户可以在每次迭代中按自己的需要随意改变 `COUNT` 值， 只要记得将上次迭代返回的游标用到下次迭代里面就可以了。

MATCH 选项

和 *KEYS* 命令一样， 增量式迭代命令也可以通过提供一个 glob 风格的模式参数， 让命令只返回和给定模式相匹配的元素， 这一点可以通过在执行增量式迭代命令时， 通过给定 `MATCH <pattern>` 参数来实现。

以下是一个使用 `MATCH` 选项进行迭代的示例：

```
redis 127.0.0.1:6379> sadd myset 1 2 3 foo foobar feelsgood
(integer) 6

redis 127.0.0.1:6379> sscan myset 0 match f*
1) "0"
2) 1) "foo"
   2) "feelsgood"
   3) "foobar"
```

需要注意的是， 对元素的模式匹配工作是在命令从数据集中取出元素之后， 向客户端返回元素之前的这段时间内进行的， 所以如果被迭代的数据集中只有少量元素和模式相匹配， 那么迭代命令或许会在多次执行中都不返回任何元素。

以下是这种情况的一个例子：

```
redis 127.0.0.1:6379> scan 0 MATCH *11*
1) "288"
2) 1) "key:911"

redis 127.0.0.1:6379> scan 288 MATCH *11*
1) "224"
2) (empty list or set)

redis 127.0.0.1:6379> scan 224 MATCH *11*
1) "80"
2) (empty list or set)

redis 127.0.0.1:6379> scan 80 MATCH *11*
1) "176"
2) (empty list or set)

redis 127.0.0.1:6379> scan 176 MATCH *11* COUNT 1000
1) "0"
2)  1) "key:611"
    2) "key:711"
    3) "key:118"
    4) "key:117"
    5) "key:311"
    6) "key:112"
    7) "key:111"
    8) "key:110"
    9) "key:113"
   10) "key:211"
   11) "key:411"
   12) "key:115"
   13) "key:116"
   14) "key:114"
   15) "key:119"
   16) "key:811"
   17) "key:511"
   18) "key:11"
```

如你所见， 以上的大部分迭代都不返回任何元素。

在最后一次迭代， 我们通过将 `COUNT` 选项的参数设置为 `1000` ， 强制命令为本次迭代扫描更多元素， 从而使得命令返回的元素也变多了。

并发执行多个迭代

在同一时间， 可以有任意多个客户端对同一数据集进行迭代， 客户端每次执行迭代都需要传入一个游标， 并在迭代执行之后获得一个新的游标， 而这个游标就包含了迭代的所有状态， 因此， 服务器无须为迭代记录任何状态。

中途停止迭代

因为迭代的所有状态都保存在游标里面， 而服务器无须为迭代保存任何状态， 所以客户端可以在中途停止一个迭代， 而无须对服务器进行任何通知。

即使有任意数量的迭代在中途停止， 也不会产生任何问题。

使用错误的游标进行增量式迭代

使用间断的（broken）、负数、超出范围或者其他非正常的游标来执行增量式迭代并不会造成服务器崩溃， 但可能会让命令产生未定义的行为。

未定义行为指的是， 增量式命令对返回值所做的保证可能会不再为真。

只有两种游标是合法的：

1. 在开始一个新的迭代时， 游标必须为 `0` 。
2. 增量式迭代命令在执行之后返回的， 用于延续（continue）迭代过程的游标。

迭代终结的保证

增量式迭代命令所使用的算法只保证在数据集的大小有界（bounded）的情况下， 迭代才会停止， 换句话说， 如果被迭代数据集的大小不断地增长的话， 增量式迭代命令可能永远也无法完成一次完整迭代。

从直觉上可以看出， 当一个数据集不断地变大时， 想要访问这个数据集中的所有元素就需要做越来越多的工作， 能否结束一个迭代取决于用户执行迭代的速度是否比数据集增长的速度更快。

可用版本：

> >= 2.8.0

时间复杂度：

> 增量式迭代命令每次执行的复杂度为 O(1) ， 对数据集进行一次完整迭代的复杂度为 O(N) ， 其中 N 为数据集中的元素数量。

返回值：

> *SCAN* 命令、 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都返回一个包含两个元素的 multi-bulk 回复： 回复的第一个元素是字符串表示的无符号 64 位整数（游标）， 回复的第二个元素是另一个 multi-bulk 回复， 这个 multi-bulk 回复包含了本次被迭代的元素。 > > *SCAN* 命令返回的每个元素都是一个数据库键。 > > *SSCAN* 命令返回的每个元素都是一个集合成员。 > > *HSCAN* 命令返回的每个元素都是一个键值对，一个键值对由一个键和一个值组成。 > > *ZSCAN* 命令返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员（member）和一个分值（score）组成。

String（字符串）

APPEND

APPEND key value

如果 `key` 已经存在并且是一个字符串， APPEND 命令将 `value` 追加到 `key` 原来的值的末尾。

如果 `key` 不存在， APPEND 就简单地将给定 `key` 设为 `value` ，就像执行 `SET key value` 一样。

可用版本：

>= 2.0.0

时间复杂度：

平摊O(1)

返回值：

追加 `value` 之后， `key` 中字符串的长度。

```
# 对不存在的 key 执行 APPEND

redis> EXISTS myphone                  # 确保 myphone 不存在
(integer) 0

redis> APPEND myphone "nokia"          # 对不存在的 key 进行 APPEND ，等[
(integer) 5                            # 字符长度

# 对已存在的字符串进行 APPEND

redis> APPEND myphone " - 1110"        # 长度从 5 个字符增加到 12 个字符
(integer) 12

redis> GET myphone
"nokia - 1110"
```

模式：时间序列(Time series)

APPEND 可以为一系列定长(fixed-size)数据(sample)提供一种紧凑的表示方式，通常称之为时间序列。

每当一个新数据到达的时候，执行以下命令：

```
APPEND timeseries "fixed-size sample"
```

然后可以通过以下的方式访问时间序列的各项属性：

- *STRLEN* 给出时间序列中数据的数量
- *GETRANGE* 可以用于随机访问。只要有相关的时间信息的话，我们就可以在 Redis 2.6 中使用 Lua 脚本和 *GETRANGE* 命令实现二分查找。
- *SETRANGE* 可以用于覆盖或修改已存在的的时间序列。

这个模式的唯一缺陷是我们只能增长时间序列，而不能对时间序列进行缩短，因为 Redis 目前还没有对字符串进行修剪(tirm)的命令，但是，不管怎么说，这个模式的储存方式还是可以节省下大量的空间。

Note

可以考虑使用 UNIX 时间戳作为时间序列的键名，这样一来，可以避免单个 `key` 因为保存过大的时间序列而占用大量内存，另一方面，也可以节省下大量命名空间。

下面是一个时间序列的例子：

```
redis> APPEND ts "0043"
(integer) 4

redis> APPEND ts "0035"
(integer) 8

redis> GETRANGE ts 0 3
"0043"

redis> GETRANGE ts 4 7
"0035"
```

BITCOUNT

BITCOUNT key [start] [end]

计算给定字符串中，被设置为 `1` 的比特位的数量。

一般情况下，给定的整个字符串都会被进行计数，通过指定额外的 `start` 或 `end` 参数，可以让计数只在特定的位上进行。

`start` 和 `end` 参数的设置和 *GETRANGE* 命令类似，都可以使用负数值：比如 `-1` 表示最后一个字节， `-2` 表示倒数第二个字节，以此类推。

不存在的 `key` 被当成是空字符串来处理，因此对一个不存在的 `key` 进行 `BITCOUNT` 操作，结果为 `0` 。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

被设置为 `1` 的位的数量。

```
redis> BITCOUNT bits
(integer) 0

redis> SETBIT bits 0 1          # 0001
(integer) 0

redis> BITCOUNT bits
(integer) 1

redis> SETBIT bits 3 1          # 1001
(integer) 0

redis> BITCOUNT bits
(integer) 2
```

模式：使用 bitmap 实现用户上线次数统计

Bitmap 对于一些特定类型的计算非常有效。

假设现在我们希望记录自己网站上的用户的上线频率，比如说，计算用户 A 上线了多少天，用户 B 上线了多少天，诸如此类，以此作为数据，从而决定让哪些用户参加 beta 测试等活动 —— 这个模式可以使用 *SETBIT* 和 *BITCOUNT* 来实现。

比如说，每当用户在某一天上线的时候，我们就使用 *SETBIT*，以用户名作为 `key`，将那天所代表的网站的上线日作为 `offset` 参数，并将这个 `offset` 上的为设置为 `1`。

举个例子，如果今天是网站上线的第 100 天，而用户 peter 在今天阅览过网站，那么执行命令 `SETBIT peter 100 1`；如果明天 peter 也继续阅览网站，那么执行命令 `SETBIT peter 101 1`，以此类推。

当要计算 peter 总共以来的上线次数时，就使用 *BITCOUNT* 命令：执行 `BITCOUNT peter`，得出的结果就是 peter 上线的总天数。

更详细的实现可以参考博文(墙外) Fast, easy, realtime metrics using Redis bitmaps。

性能

前面的上线次数统计例子，即使运行 10 年，占用的空间也只是每个用户 10*365 比特位(bit)，也即是每个用户 456 字节。对于这种大小的数据来说，*BITCOUNT* 的处理速度就像 *GET* 和 *INCR* 这种 O(1) 复杂度的操作一样快。

如果你的 bitmap 数据非常大，那么可以考虑使用以下两种方法：

1. 将一个大的 bitmap 分散到不同的 key 中，作为小的 bitmap 来处理。使用 Lua 脚本可以很方便地完成这一工作。
2. 使用 *BITCOUNT* 的 `start` 和 `end` 参数，每次只对所需的部分位进行计算，将位的累积工作(accumulating)放到客户端进行，并且对结果进行缓存(caching)。

BITOP

BITOP operation destkey key [key ...]

对一个或多个保存二进制位的字符串 `key` 进行位元操作，并将结果保存到 `destkey` 上。

`operation` 可以是 `AND` 、 `OR` 、 `NOT` 、 `XOR` 这四种操作中的任意一种：

- `BITOP AND destkey key [key ...]` ，对一个或多个 `key` 求逻辑并，并将结果保存到 `destkey` 。
- `BITOP OR destkey key [key ...]` ，对一个或多个 `key` 求逻辑或，并将结果保存到 `destkey` 。
- `BITOP XOR destkey key [key ...]` ，对一个或多个 `key` 求逻辑异或，并将结果保存到 `destkey` 。
- `BITOP NOT destkey key` ，对给定 `key` 求逻辑非，并将结果保存到 `destkey` 。

除了 `NOT` 操作之外，其他操作都可以接受一个或多个 `key` 作为输入。

处理不同长度的字符串

当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 `0` 。

空的 `key` 也被看作是包含 `0` 的字符串序列。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

保存到 `destkey` 的字符串的长度，和输入 `key` 中最长的字符串长度相等。

Note

BITOP 的复杂度为 O(N) ，当处理大型矩阵(matrix)或者进行大数据量的统计时，最好将任务指派到附属节点(slave)进行，避免阻塞主节点。

```
redis> SETBIT bits-1 0 1        # bits-1 = 1001
(integer) 0

redis> SETBIT bits-1 3 1
(integer) 0

redis> SETBIT bits-2 0 1        # bits-2 = 1011
(integer) 0

redis> SETBIT bits-2 1 1
(integer) 0

redis> SETBIT bits-2 3 1
(integer) 0

redis> BITOP AND and-result bits-1 bits-2
(integer) 1

redis> GETBIT and-result 0      # and-result = 1001
(integer) 1

redis> GETBIT and-result 1
(integer) 0

redis> GETBIT and-result 2
(integer) 0

redis> GETBIT and-result 3
(integer) 1
```

DECR

DECR key

将 `key` 中储存的数字值减一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 DECR 命令之后 `key` 的值。

```
# 对存在的数字值 key 进行 DECR

redis> SET failure_times 10
OK

redis> DECR failure_times
(integer) 9

# 对不存在的 key 值进行 DECR

redis> EXISTS count
(integer) 0

redis> DECR count
(integer) -1

# 对存在但不是数值的 key 进行 DECR

redis> SET company YOUR_CODE_SUCKS.LLC
OK

redis> DECR company
(error) ERR value is not an integer or out of range
```

DECRBY

DECRBY key decrement

将 `key` 所储存的值减去减量 `decrement` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECRBY 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于更多递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

减去 `decrement` 之后， `key` 的值。

```
# 对已存在的 key 进行 DECRBY

redis> SET count 100
OK

redis> DECRBY count 20
(integer) 80

# 对不存在的 key 进行DECRBY

redis> EXISTS pages
(integer) 0

redis> DECRBY pages 10
(integer) -10
```

GET

GET key

返回 `key` 所关联的字符串值。

如果 `key` 不存在那么返回特殊值 `nil` 。

假如 `key` 储存的值不是字符串类型，返回一个错误，因为 GET 只能用于处理字符串值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `nil` ，否则，返回 `key` 的值。如果 `key` 不是字符串类型，那么返回一个错误。

```
# 对不存在的 key 或字符串类型 key 进行 GET

redis> GET db
(nil)

redis> SET db redis
OK

redis> GET db
"redis"

# 对不是字符串类型的 key 进行 GET

redis> DEL db
(integer) 1

redis> LPUSH db redis mongodb mysql
(integer) 3

redis> GET db
(error) ERR Operation against a key holding the wrong kind of value
```

GETBIT

GETBIT key offset

对 `key` 所储存的字符串值，获取指定偏移量上的位(bit)。

当 `offset` 比字符串值的长度大，或者 `key` 不存在时，返回 `0` 。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

字符串值指定偏移量上的位(bit)。

```
# 对不存在的 key 或者不存在的 offset 进行 GETBIT， 返回 0

redis> EXISTS bit
(integer) 0

redis> GETBIT bit 10086
(integer) 0

# 对已存在的 offset 进行 GETBIT

redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1
```

GETRANGE

GETRANGE key start end

返回 `key` 中字符串值的子字符串，字符串的截取范围由 `start` 和 `end` 两个偏移量决定(包括 `start` 和 `end` 在内)。

负数偏移量表示从字符串最后开始计数， `-1` 表示最后一个字符， `-2` 表示倒数第二个，以此类推。

GETRANGE 通过保证子字符串的值域(range)不超过实际字符串的值域来处理超出范围的值域请求。

Note

在 <= 2.0 的版本里，GETRANGE 被叫作 SUBSTR。

可用版本：

>= 2.4.0

时间复杂度：

O(N)， `N` 为要返回的字符串的长度。复杂度最终由字符串的返回值长度决定，但因为从已有字符串中取出子字符串的操作非常廉价(cheap)，所以对于长度不大的字符串，该操作的复杂度也可看作O(1)。

返回值：

截取得出的子字符串。

```
redis> SET greeting "hello, my friend"
OK

redis> GETRANGE greeting 0 4            # 返回索引0-4的字符，包括4。
"hello"

redis> GETRANGE greeting -1 -5          # 不支持回绕操作
""

redis> GETRANGE greeting -3 -1          # 负数索引
"end"

redis> GETRANGE greeting 0 -1           # 从第一个到最后一个
"hello, my friend"

redis> GETRANGE greeting 0 1008611      # 值域范围不超过实际字符串，超过部
"hello, my friend"
```

GETSET

GETSET key value

将给定 `key` 的值设为 `value` ，并返回 `key` 的旧值(old value)。

当 `key` 存在但不是字符串类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

返回给定 `key` 的旧值。当 `key` 没有旧值时，也即是， `key` 不存在时，返回 `nil` 。

```
redis> GETSET db mongodb    # 没有旧值，返回 nil
(nil)

redis> GET db
"mongodb"

redis> GETSET db redis      # 返回旧值 mongodb
"mongodb"

redis> GET db
"redis"
```

模式

GETSET 可以和 *INCR* 组合使用，实现一个有原子性(atomic)复位操作的计数器(counter)。

举例来说，每次当某个事件发生时，进程可能对一个名为 `mycount` 的 `key` 调用 *INCR* 操作，通常我们还要在一个原子时间内同时完成获得计数器的值和将计数器值复位为 `0` 两个操作。

可以用命令 `GETSET mycounter 0` 来实现这一目标。

```
redis> INCR mycount
(integer) 11

redis> GETSET mycount 0  # 一个原子内完成 GET mycount 和 SET mycount
"11"

redis> GET mycount       # 计数器被重置
"0"
```

INCR

INCR key

将 `key` 中储存的数字值增一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

Note

这是一个针对字符串的操作，因为 Redis 没有专用的整数类型，所以 key 内储存的字符串被解释为十进制 64 位有符号整数来执行 INCR 操作。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 INCR 命令之后 `key` 的值。

```
redis> SET page_view 20
OK

redis> INCR page_view
(integer) 21

redis> GET page_view    # 数字值在 Redis 中以字符串的形式保存
"21"
```

模式：计数器

计数器是 Redis 的原子性自增操作可实现的最直观的模式了，它的想法相当简单：每当某个操作发生时，向 Redis 发送一个 INCR 命令。

比如在一个 web 应用程序中，如果想知道用户在一年中每天的点击量，那么只要将用户 ID 以及相关的日期信息作为键，并在每次用户点击页面时，执行一次自增操作即可。

比如用户名是 `peter` ，点击时间是 2012 年 3 月 22 日，那么执行命令 `INCR peter::2012.3.22` 。

可以用以下几种方式扩展这个简单的模式：

- 可以通过组合使用 INCR 和 *EXPIRE* ，来达到只在规定的生存时间内进行计数(counting)的目的。
- 客户端可以通过使用 *GETSET* 命令原子性地获取计数器的当前值并将计数器清零，更多信息请参考 *GETSET* 命令。
- 使用其他自增/自减操作，比如 *DECR* 和 *INCRBY* ，用户可以通过执行不同的操作增加或减少计数器的值，比如在游戏中的记分器就可能用到这些命令。

模式：限速器

限速器是特殊化的计算器，它用于限制一个操作可以被执行的速率(rate)。

限速器的典型用法是限制公开 API 的请求次数，以下是一个限速器实现示例，它将 API 的最大请求数限制在每个 IP 地址每秒钟十个之内：

```
FUNCTION LIMIT_API_CALL(ip)
ts = CURRENT_UNIX_TIME()
keyname = ip+":"+ts
current = GET(keyname)

IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
END

IF current == NULL THEN
    MULTI
        INCR(keyname, 1)
        EXPIRE(keyname, 1)
    EXEC
ELSE
    INCR(keyname, 1)
END

PERFORM_API_CALL()
```

这个实现每秒钟为每个 IP 地址使用一个不同的计数器，并用 *EXPIRE* 命令设置生存时间(这样 Redis 就会负责自动删除过期的计数器)。

注意，我们使用事务打包执行 *INCR* 命令和 *EXPIRE* 命令，避免引入竞争条件，保证每次调用 API 时都可以正确地对计数器进行自增操作并设置生存时间。

以下是另一个限速器实现：

```
FUNCTION LIMIT_API_CALL(ip):
current = GET(ip)
IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
ELSE
    value = INCR(ip)
    IF value == 1 THEN
        EXPIRE(ip,1)
    END
    PERFORM_API_CALL()
END
```

这个限速器只使用单个计数器，它的生存时间为一秒钟，如果在一秒钟内，这个计数器的值大于 `10` 的话，那么访问就会被禁止。

这个新的限速器在思路方面是没有问题的，但它在实现方面不够严谨，如果我们仔细观察一下的话，就会发现在 *INCR* 和 *EXPIRE* 之间存在着一个竞争条件，假如客户端在执行 *INCR* 之后，因为某些原因(比如客户端失败)而忘记设置 *EXPIRE* 的话，那么这个计数器就会一直存在下去，造成每个用户只能访问 `10` 次，噢，这简直是个灾难！

要消灭这个实现中的竞争条件，我们可以将它转化为一个 Lua 脚本，并放到 Redis 中运行(这个方法仅限于 Redis 2.6 及以上的版本)：

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

通过将计数器作为脚本放到 Redis 上运行，我们保证了 *INCR* 和 *EXPIRE* 两个操作的原子性，现在这个脚本实现不会引入竞争条件，它可以运作的很好。

关于在 Redis 中运行 Lua 脚本的更多信息，请参考 *EVAL* 命令。

还有另一种消灭竞争条件的方法，就是使用 Redis 的列表结构来代替 *INCR* 命令，这个方法无须脚本支持，因此它在 Redis 2.6 以下的版本也可以运行得很好：

```
FUNCTION LIMIT_API_CALL(ip)
current = LLEN(ip)
IF current > 10 THEN
    ERROR "too many requests per second"
ELSE
    IF EXISTS(ip) == FALSE
        MULTI
            RPUSH(ip,ip)
            EXPIRE(ip,1)
        EXEC
    ELSE
        RPUSHX(ip,ip)
    END
    PERFORM_API_CALL()
END
```

新的限速器使用了列表结构作为容器， *LLEN* 用于对访问次数进行检查，一个事务包裹着 *RPUSH* 和 *EXPIRE* 两个命令，用于在第一次执行计数时创建列表，并正确设置地设置过期时间，最后， *RPUSHX* 在后续的计数操作中进行增加操作。

INCRBY

INCRBY key increment

将 `key` 所储存的值加上增量 `increment` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCRBY 命令。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

加上 `increment` 之后， `key` 的值。

```
# key 存在且是数字值

redis> SET rank 50
OK

redis> INCRBY rank 20
(integer) 70

redis> GET rank
"70"

# key 不存在时

redis> EXISTS counter
(integer) 0

redis> INCRBY counter 30
(integer) 30

redis> GET counter
"30"

# key 不是数字值时

redis> SET book "long long ago..."
OK

redis> INCRBY book 200
(error) ERR value is not an integer or out of range
```

INCRBYFLOAT

INCRBYFLOAT key increment

为 `key` 中所储存的值加上浮点数增量 `increment` 。

如果 `key` 不存在，那么 INCRBYFLOAT 会先将 `key` 的值设为 `0` ，再执行加法操作。

如果命令执行成功，那么 `key` 的值会被更新为（执行加法之后的）新值，并且新值会以字符串的形式返回给调用者。

无论是 `key` 的值，还是增量 `increment` ，都可以使用像 `2.0e7` 、 `3e5` 、 `90e-2` 那样的指数符号(exponential notation)来表示，但是，执行 **INCRBYFLOAT** 命令之后的值总是以同样的形式储存，也即是，它们总是由一个数字，一个（可选的）小数点和一个任意位的小数部分组成（比如 `3.14` 、 `69.768` ，诸如此类)，小数部分尾随的 `0` 会被移除，如果有需要的话，还会将浮点数改为整数（比如 `3.0` 会被保存成 `3` ）。

除此之外，无论加法计算所得的浮点数的实际精度有多长， INCRBYFLOAT 的计算结果也最多只能表示小数点的后十七位。

当以下任意一个条件发生时，返回一个错误：

- `key` 的值不是字符串类型(因为 Redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- `key` 当前的值或者给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number）

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行命令之后 `key` 的值。

```
# 值和增量都不是指数符号

redis> SET mykey 10.50
OK

redis> INCRBYFLOAT mykey 0.1
"10.6"

# 值和增量都是指数符号

redis> SET mykey 314e-2
OK

redis> GET mykey                # 用 SET 设置的值可以是指数符号
"314e-2"

redis> INCRBYFLOAT mykey 0      # 但执行 INCRBYFLOAT 之后格式会被改成非
"3.14"

# 可以对整数类型执行

redis> SET mykey 3
OK

redis> INCRBYFLOAT mykey 1.1
"4.1"

# 后跟的 0 会被移除

redis> SET mykey 3.0
OK

redis> GET mykey                                    # SET 设置的值小
"3.0"

redis> INCRBYFLOAT mykey 1.000000000000000000000    # 但 INCRBYFLOA
"4"

redis> GET mykey
"4"
```

MGET

MGET key [key ...]

返回所有(一个或多个)给定 `key` 的值。

如果给定的 `key` 里面，有某个 `key` 不存在，那么这个 `key` 返回特殊值 `nil` 。因此，该命令永不失败。

可用版本：

>= 1.0.0

时间复杂度:

O(N) , `N` 为给定 `key` 的数量。

返回值：

一个包含所有给定 `key` 的值的列表。

```
redis> SET redis redis.com
OK

redis> SET mongodb mongodb.org
OK

redis> MGET redis mongodb
1) "redis.com"
2) "mongodb.org"

redis> MGET redis mongodb mysql     # 不存在的 mysql 返回 nil
1) "redis.com"
2) "mongodb.org"
3) (nil)
```

MSET

MSET key value [key value ...]

同时设置一个或多个 `key-value` 对。

如果某个给定 `key` 已经存在，那么 MSET 会用新值覆盖原来的旧值，如果这不是你所希望的效果，请考虑使用 *MSETNX* 命令：它只会在所有给定 `key` 都不存在的情况下进行设置操作。

MSET 是一个原子性(atomic)操作，所有给定 `key` 都会在同一时间内被设置，某些给定 `key` 被更新而另一些给定 `key` 没有改变的情况，不可能发生。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 数量。

返回值：

总是返回 `OK` (因为 `MSET` 不可能失败)

```
redis> MSET date "2012.3.30" time "11:00 a.m." weather "sunny"
OK

redis> MGET date time weather
1) "2012.3.30"
2) "11:00 a.m."
3) "sunny"

# MSET 覆盖旧值例子

redis> SET google "google.hk"
OK

redis> MSET google "google.com"
OK

redis> GET google
"google.com"
```

MSETNX

MSETNX key value [key value ...]

同时设置一个或多个 `key-value` 对，当且仅当所有给定 `key` 都不存在。

即使只有一个给定 `key` 已存在， MSETNX 也会拒绝执行所有给定 `key` 的设置操作。

MSETNX 是原子性的，因此它可以用作设置多个不同 `key` 表示不同字段(field)的唯一性逻辑对象(unique logic object)，所有字段要么全被设置，要么全不被设置。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 的数量。

返回值：

当所有 `key` 都成功设置，返回 `1` 。如果所有给定 `key` 都设置失败(至少有一个 `key` 已经存在)，那么返回 `0` 。

```
# 对不存在的 key 进行 MSETNX

redis> MSETNX rmdbs "MySQL" nosql "MongoDB" key-value-store "redis'
(integer) 1

redis> MGET rmdbs nosql key-value-store
1) "MySQL"
2) "MongoDB"
3) "redis"

# MSET 的给定 key 当中有已存在的 key

redis> MSETNX rmdbs "Sqlite" language "python"  # rmdbs 键已经存在，扌
(integer) 0

redis> EXISTS language                          # 因为 MSET 是原子性扌
(integer) 0

redis> GET rmdbs                                # rmdbs 也没有被修改
"MySQL"
```

PSETEX

PSETEX key milliseconds value

这个命令和 *SETEX* 命令相似，但它以毫秒为单位设置 `key` 的生存时间，而不是像 *SETEX* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。

```
redis> PSETEX mykey 1000 "Hello"
OK

redis> PTTL mykey
(integer) 999

redis> GET mykey
"Hello"
```

SET

SET key value [EX seconds] [PX milliseconds] [NX|XX]

将字符串值 `value` 关联到 `key` 。

如果 `key` 已经持有其他值， SET 就覆写旧值，无视类型。

对于某个原本带有生存时间（TTL）的键来说， 当 *SET* 命令成功在这个键上执行时， 这个键原有的 TTL 将被清除。

可选参数

从 Redis 2.6.12 版本开始， *SET* 命令的行为可以通过一系列参数来修改：

- `EX second` ：设置键的过期时间为 `second` 秒。 `SET key value EX second` 效果等同于 `SETEX key second value` 。
- `PX millisecond` ：设置键的过期时间为 `millisecond` 毫秒。 `SET key value PX millisecond` 效果等同于 `PSETEX key millisecond value` 。
- `NX` ：只在键不存在时，才对键进行设置操作。 `SET key value NX` 效果等同于 `SETNX key value` 。
- `XX` ：只在键已经存在时，才对键进行设置操作。

Note

因为 *SET* 命令可以通过参数来实现和 *SETNX* 、 *SETEX* 和 *PSETEX* 三个命令的效果，所以将来的 Redis 版本可能会废弃并最终移除 *SETNX* 、 *SETEX* 和 *PSETEX* 这三个命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

在 Redis 2.6.12 版本以前， *SET* 命令总是返回 `OK` 。

从 Redis 2.6.12 版本开始， *SET* 在设置操作成功完成时，才返回 `OK` 。如果设置了 `NX` 或者 `XX` ，但因为条件没达到而造成设置操作未执行，那么命令返回空批量回复（NULL Bulk Reply）。

```
# 对不存在的键进行设置

redis 127.0.0.1:6379> SET key "value"
```

```
OK

redis 127.0.0.1:6379> GET key
"value"

# 对已存在的键进行设置

redis 127.0.0.1:6379> SET key "new-value"
OK

redis 127.0.0.1:6379> GET key
"new-value"

# 使用 EX 选项

redis 127.0.0.1:6379> SET key-with-expire-time "hello" EX 10086
OK

redis 127.0.0.1:6379> GET key-with-expire-time
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-time
(integer) 10069

# 使用 PX 选项

redis 127.0.0.1:6379> SET key-with-pexpire-time "moto" PX 123321
OK

redis 127.0.0.1:6379> GET key-with-pexpire-time
"moto"

redis 127.0.0.1:6379> PTTL key-with-pexpire-time
(integer) 111939

# 使用 NX 选项

redis 127.0.0.1:6379> SET not-exists-key "value" NX
OK      # 键不存在，设置成功

redis 127.0.0.1:6379> GET not-exists-key
"value"

redis 127.0.0.1:6379> SET not-exists-key "new-value" NX
(nil)   # 键已经存在，设置失败

redis 127.0.0.1:6379> GEt not-exists-key
"value" # 维持原值不变

# 使用 XX 选项

redis 127.0.0.1:6379> EXISTS exists-key
(integer) 0
```

```
redis 127.0.0.1:6379> SET exists-key "value" XX
(nil)   # 因为键不存在，设置失败

redis 127.0.0.1:6379> SET exists-key "value"
OK      # 先给键设置一个值

redis 127.0.0.1:6379> SET exists-key "new-value" XX
OK      # 设置新值成功

redis 127.0.0.1:6379> GET exists-key
"new-value"

# NX 或 XX 可以和 EX 或者 PX 组合使用

redis 127.0.0.1:6379> SET key-with-expire-and-NX "hello" EX 10086 N
OK

redis 127.0.0.1:6379> GET key-with-expire-and-NX
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-and-NX
(integer) 10063

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "old value"
OK

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "new value" PX 12
OK

redis 127.0.0.1:6379> GET key-with-pexpire-and-XX
"new value"

redis 127.0.0.1:6379> PTTL key-with-pexpire-and-XX
(integer) 112999

# EX 和 PX 可以同时出现，但后面给出的选项会覆盖前面给出的选项

redis 127.0.0.1:6379> SET key "value" EX 1000 PX 5000000
OK

redis 127.0.0.1:6379> TTL key
(integer) 4993  # 这是 PX 参数设置的值

redis 127.0.0.1:6379> SET another-key "value" PX 5000000 EX 1000
OK

redis 127.0.0.1:6379> TTL another-key
(integer) 997   # 这是 EX 参数设置的值
```

使用模式

命令 `SET resource-name anystring NX EX max-lock-time` 是一种在 Redis 中实现锁的简单方法。

客户端执行以上的命令：

- 如果服务器返回 `OK` ，那么这个客户端获得锁。
- 如果服务器返回 `NIL` ，那么客户端获取锁失败，可以在稍后再重试。

设置的过期时间到达之后，锁将自动释放。

可以通过以下修改，让这个锁实现更健壮：

- 不使用固定的字符串作为键的值，而是设置一个不可猜测（non-guessable）的长随机字符串，作为口令串（token）。
- 不使用 *DEL* 命令来释放锁，而是发送一个 Lua 脚本，这个脚本只在客户端传入的值和键的口令串相匹配时，才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

这个脚本可以通过 `EVAL ...script... 1 resource-name token-value` 命令来调用。

SETBIT

SETBIT key offset value

对 `key` 所储存的字符串值，设置或清除指定偏移量上的位(bit)。

位的设置或清除取决于 `value` 参数，可以是 `0` 也可以是 `1` 。

当 `key` 不存在时，自动生成一个新的字符串值。

字符串会进行伸展(grown)以确保它可以将 `value` 保存在指定的偏移量上。当字符串值进行伸展时，空白位置以 `0` 填充。

`offset` 参数必须大于或等于 `0` ，小于 2^32 (bit 映射被限制在 512 MB 之内)。

Warning

对使用大的 `offset` 的 SETBIT 操作来说，内存分配可能造成 Redis 服务器被阻塞。具体参考 *SETRANGE* 命令，warning(警告)部分。

可用版本：

>= 2.2.0

时间复杂度:

O(1)

返回值：

指定偏移量原来储存的位。

```
redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1

redis> GETBIT bit 100   # bit 默认被初始化为 0
(integer) 0
```

SETEX

SETEX key seconds value

将值 `value` 关联到 `key` ，并将 `key` 的生存时间设为 `seconds` (以秒为单位)。

如果 `key` 已经存在， SETEX 命令将覆写旧值。

这个命令类似于以下两个命令：

```
SET key value
EXPIRE key seconds  # 设置生存时间
```

不同之处是， SETEX 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成，该命令在 Redis 用作缓存时，非常实用。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。当 `seconds` 参数不合法时，返回一个错误。

```
# 在 key 不存在时进行 SETEX

redis> SETEX cache_user_id 60 10086
OK

redis> GET cache_user_id  # 值
"10086"

redis> TTL cache_user_id  # 剩余生存时间
(integer) 49

# key 已经存在时，SETEX 覆盖旧值

redis> SET cd "timeless"
OK

redis> SETEX cd 3000 "goodbye my love"
OK

redis> GET cd
"goodbye my love"

redis> TTL cd
(integer) 2997
```

SETNX

SETNX key value

将 `key` 的值设为 `value` ，当且仅当 `key` 不存在。

若给定的 `key` 已经存在，则 SETNX 不做任何动作。

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。设置失败，返回 `0` 。

```
redis> EXISTS job                # job 不存在
(integer) 0

redis> SETNX job "programmer"    # job 设置成功
(integer) 1

redis> SETNX job "code-farmer"   # 尝试覆盖 job ，失败
(integer) 0

redis> GET job                   # 没有被覆盖
"programmer"
```

SETRANGE

SETRANGE key offset value

用 `value` 参数覆写(overwrite)给定 `key` 所储存的字符串值，从偏移量 `offset` 开始。

不存在的 `key` 当作空白字符串处理。

SETRANGE 命令会确保字符串足够长以便将 `value` 设置在指定的偏移量上，如果给定 `key` 原来储存的字符串长度比偏移量小(比如字符串只有 `5` 个字符长，但你设置的 `offset` 是 `10`)，那么原字符和偏移量之间的空白将用零字节(zerobytes, `"\x00"`)来填充。

注意你能使用的最大偏移量是 2^29-1(536870911) ，因为 Redis 字符串的大小被限制在 512 兆(megabytes)以内。如果你需要使用比这更大的空间，你可以使用多个 `key` 。

Warning

当生成一个很长的字符串时，Redis 需要分配内存空间，该操作有时候可能会造成服务器阻塞(block)。在2010年的Macbook Pro上，设置偏移量为 536870911(512MB 内存分配)，耗费约 300 毫秒， 设置偏移量为 134217728(128MB 内存分配)，耗费约 80 毫秒，设置偏移量 33554432(32MB 内存分配)，耗费约 30 毫秒，设置偏移量为 8388608(8MB 内存分配)，耗费约 8 毫秒。 注意若首次内存分配成功之后，再对同一个 `key` 调用 SETRANGE 操作，无须再重新内存。

可用版本：

>= 2.2.0

时间复杂度：

对小(small)的字符串，平摊复杂度O(1)。(关于什么字符串是”小”的，请参考 *APPEND* 命令)否则为O(M)， `M` 为 `value` 参数的长度。

返回值：

被 SETRANGE 修改之后，字符串的长度。

```
# 对非空字符串进行 SETRANGE

redis> SET greeting "hello world"
OK

redis> SETRANGE greeting 6 "Redis"
(integer) 11

redis> GET greeting
"hello Redis"

# 对空字符串/不存在的 key 进行 SETRANGE

redis> EXISTS empty_string
(integer) 0

redis> SETRANGE empty_string 5 "Redis!"   # 对不存在的 key 使用 SETRA
(integer) 11

redis> GET empty_string                   # 空白处被"\x00"填充
"\x00\x00\x00\x00\x00Redis!"
```

模式

因为有了 SETRANGE 和 *GETRANGE* 命令，你可以将 Redis 字符串用作具有O(1)随机访问时间的线性数组，这在很多真实用例中都是非常快速且高效的储存方式，具体请参考 *APPEND* 命令的『模式：时间序列』部分。

STRLEN

STRLEN key

返回 `key` 所储存的字符串值的长度。

当 `key` 储存的不是字符串值时，返回一个错误。

可用版本：

>= 2.2.0

复杂度：

O(1)

返回值：

字符串值的长度。当 `key` 不存在时，返回 `0` 。

```
# 获取字符串的长度

redis> SET mykey "Hello world"
OK

redis> STRLEN mykey
(integer) 11

# 不存在的 key 长度为 0

redis> STRLEN nonexisting
(integer) 0
```

Hash（哈希表）

HDEL

HDEL key field [field ...]

删除哈希表 `key` 中的一个或多个指定域，不存在的域将被忽略。

Note

在Redis2.4以下的版本里， HDEL 每次只能删除单个域，如果你需要在一个原子时间内删除多个域，请将命令包含在 *MULTI* / *EXEC* 块内。

可用版本：

>= 2.0.0

时间复杂度:

O(N)， `N` 为要删除的域的数量。

返回值:

被成功移除的域的数量，不包括被忽略的域。

```
# 测试数据

redis> HGETALL abbr
1) "a"
2) "apple"
3) "b"
4) "banana"
5) "c"
6) "cat"
7) "d"
8) "dog"

# 删除单个域

redis> HDEL abbr a
(integer) 1

# 删除不存在的域

redis> HDEL abbr not-exists-field
(integer) 0

# 删除多个域

redis> HDEL abbr b c
(integer) 2

redis> HGETALL abbr
1) "d"
2) "dog"
```

HEXISTS

HEXISTS key field

查看哈希表 `key` 中，给定域 `field` 是否存在。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果哈希表含有给定域，返回 `1` 。如果哈希表不含有给定域，或 `key` 不存在，返回 `0` 。

```
redis> HEXISTS phone myphone
(integer) 0

redis> HSET phone myphone nokia-1110
(integer) 1

redis> HEXISTS phone myphone
(integer) 1
```

HGET

HGET key field

返回哈希表 `key` 中给定域 `field` 的值。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

给定域的值。当给定域不存在或是给定 `key` 不存在时，返回 `nil` 。

```
# 域存在

redis> HSET site redis redis.com
(integer) 1

redis> HGET site redis
"redis.com"

# 域不存在

redis> HGET site mysql
(nil)
```

HGETALL

HGETALL key

返回哈希表 `key` 中，所有的域和值。

在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

以列表形式返回哈希表的域和域的值。若 `key` 不存在，返回空列表。

```
redis> HSET people jack "Jack Sparrow"
(integer) 1

redis> HSET people gump "Forrest Gump"
(integer) 1

redis> HGETALL people
1) "jack"          # 域
2) "Jack Sparrow"  # 值
3) "gump"
4) "Forrest Gump"
```

HINCRBY

HINCRBY key field increment

为哈希表 `key` 中的域 `field` 的值加上增量 `increment` 。

增量也可以为负数，相当于对给定域进行减法操作。

如果 `key` 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。

如果域 `field` 不存在，那么在执行命令前，域的值被初始化为 `0` 。

对一个储存字符串值的域 `field` 执行 HINCRBY 命令将造成一个错误。

本操作的值被限制在 64 位(bit)有符号数字表示之内。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

执行 HINCRBY 命令之后，哈希表 `key` 中域 `field` 的值。

```
# increment 为正数

redis> HEXISTS counter page_view    # 对空域进行设置
(integer) 0

redis> HINCRBY counter page_view 200
(integer) 200

redis> HGET counter page_view
"200"

# increment 为负数

redis> HGET counter page_view
"200"

redis> HINCRBY counter page_view -50
(integer) 150

redis> HGET counter page_view
"150"

# 尝试对字符串值的域执行HINCRBY命令

redis> HSET myhash string hello,world       # 设定一个字符串值
(integer) 1

redis> HGET myhash string
"hello,world"

redis> HINCRBY myhash string 1              # 命令执行失败，错误。
(error) ERR hash value is not an integer

redis> HGET myhash string                   # 原值不变
"hello,world"
```

HINCRBYFLOAT

HINCRBYFLOAT key field increment

为哈希表 `key` 中的域 `field` 加上浮点数增量 `increment` 。

如果哈希表中没有域 `field` ，那么 HINCRBYFLOAT 会先将域 `field` 的值设为 `0` ，然后再执行加法操作。

如果键 `key` 不存在，那么 HINCRBYFLOAT 会先创建一个哈希表，再创建域 `field` ，最后再执行加法操作。

当以下任意一个条件发生时，返回一个错误：

- 域 `field` 的值不是字符串类型(因为 redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型)
- 域 `field` 当前的值或给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number)

HINCRBYFLOAT 命令的详细功能和 *INCRBYFLOAT* 命令类似，请查看 *INCRBYFLOAT* 命令获取更多相关信息。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行加法操作之后 `field` 域的值。

```
# 值和增量都是普通小数

redis> HSET mykey field 10.50
(integer) 1
redis> HINCRBYFLOAT mykey field 0.1
"10.6"

# 值和增量都是指数符号

redis> HSET mykey field 5.0e3
(integer) 0
redis> HINCRBYFLOAT mykey field 2.0e2
"5200"

# 对不存在的键执行 HINCRBYFLOAT

redis> EXISTS price
(integer) 0
redis> HINCRBYFLOAT price milk 3.5
"3.5"
redis> HGETALL price
1) "milk"
2) "3.5"

# 对不存在的域进行 HINCRBYFLOAT

redis> HGETALL price
1) "milk"
2) "3.5"
redis> HINCRBYFLOAT price coffee 4.5   # 新增 coffee 域
"4.5"
redis> HGETALL price
1) "milk"
2) "3.5"
3) "coffee"
4) "4.5"
```

HKEYS

HKEYS key

返回哈希表 `key` 中的所有域。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有域的表。当 `key` 不存在时，返回一个空表。

```
# 哈希表非空

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HKEYS website
1) "google"
2) "yahoo"

# 空哈希表/key不存在

redis> EXISTS fake_key
(integer) 0

redis> HKEYS fake_key
(empty list or set)
```

HLEN

HLEN key

返回哈希表 `key` 中域的数量。

时间复杂度：

O(1)

返回值：

哈希表中域的数量。当 `key` 不存在时，返回 `0` 。

```
redis> HSET db redis redis.com
(integer) 1

redis> HSET db mysql mysql.com
(integer) 1

redis> HLEN db
(integer) 2

redis> HSET db mongodb mongodb.org
(integer) 1

redis> HLEN db
(integer) 3
```

HMGET

HMGET key field [field ...]

返回哈希表 `key` 中，一个或多个给定域的值。

如果给定的域不存在于哈希表，那么返回一个 `nil` 值。

因为不存在的 `key` 被当作一个空哈希表来处理，所以对一个不存在的 `key` 进行 HMGET 操作将返回一个只带有 `nil` 值的表。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为给定域的数量。

返回值：

一个包含多个给定域的关联值的表，表值的排列顺序和给定域参数的请求顺序一样。

```
redis> HMSET pet dog "doudou" cat "nounou"    # 一次设置多个域
OK

redis> HMGET pet dog cat fake_pet             # 返回值的顺序和传入参数
1) "doudou"
2) "nounou"
3) (nil)                                      # 不存在的域返回nil值
```

HMSET

HMSET key field value [field value ...]

同时将多个 `field-value` (域-值)对设置到哈希表 `key` 中。

此命令会覆盖哈希表中已存在的域。

如果 `key` 不存在，一个空哈希表被创建并执行 HMSET 操作。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为 `field-value` 对的数量。

返回值：

如果命令执行成功，返回 `OK` 。当 `key` 不是哈希表(hash)类型时，返回一个错误。

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HGET website google
"www.google.com"

redis> HGET website yahoo
"www.yahoo.com"
```

HSET

HSET key field value

将哈希表 `key` 中的域 `field` 的值设为 `value` 。

如果 `key` 不存在，一个新的哈希表被创建并进行 HSET 操作。

如果域 `field` 已经存在于哈希表中，旧值将被覆盖。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果 `field` 是哈希表中的一个新建域，并且值设置成功，返回 `1` 。如果哈希表中域 `field` 已经存在且旧值已被新值覆盖，返回 `0` 。

```
redis> HSET website google "www.g.cn"       # 设置一个新域
(integer) 1

redis> HSET website google "www.google.com" # 覆盖一个旧域
(integer) 0
```

HSETNX

HSETNX key field value

将哈希表 `key` 中的域 `field` 的值设置为 `value` ，当且仅当域 `field` 不存在。

若域 `field` 已经存在，该操作无效。

如果 `key` 不存在，一个新哈希表被创建并执行 HSETNX 命令。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。如果给定域已经存在且没有操作被执行，返回 `0` 。

```
redis> HSETNX nosql key-value-store redis
(integer) 1

redis> HSETNX nosql key-value-store redis       # 操作无效，域 key-va
(integer) 0
```

HVALS

HVALS key

返回哈希表 `key` 中所有域的值。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有值的表。当 `key` 不存在时，返回一个空表。

```
# 非空哈希表

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HVALS website
1) "www.google.com"
2) "www.yahoo.com"

# 空哈希表/不存在的key

redis> EXISTS not_exists
(integer) 0

redis> HVALS not_exists
(empty list or set)
```

HSCAN

HSCAN key cursor [MATCH pattern] [COUNT count]

具体信息请参考 *SCAN* 命令。

List（列表）

BLPOP

BLPOP key [key ...] timeout

BLPOP 是列表的阻塞式(blocking)弹出原语。

它是 *LPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的头元素。

非阻塞行为

当 BLPOP 被调用时，如果给定 `key` 内至少有一个非空列表，那么弹出遇到的第一个非空列表的头元素，并和被弹出元素所属的列表的名字一起，组成结果返回给调用者。

当存在多个给定 `key` 时， BLPOP 按给定 `key` 参数排列的先后顺序，依次检查各个列表。

假设现在有 `job` 、 `command` 和 `request` 三个列表，其中 `job` 不存在， `command` 和 `request` 都持有非空列表。考虑以下命令：

`BLPOP job command request 0`

BLPOP 保证返回的元素来自 `command` ，因为它是按”查找 `job` -> 查找 `command` -> 查找 `request` “这样的顺序，第一个找到的非空列表。

```
redis> DEL job command request           # 确保key都被删除
(integer) 0

redis> LPUSH command "update system..."  # 为command列表增加一个值
(integer) 1

redis> LPUSH request "visit page"        # 为request列表增加一个值
(integer) 1

redis> BLPOP job command request 0       # job 列表为空，被跳过，紧接着
1) "command"                             # 弹出元素所属的列表
2) "update system..."                    # 弹出元素所属的值
```

阻塞行为

如果所有给定 `key` 都不存在或包含空列表，那么 BLPOP 命令将阻塞连接，直到等待超时，或有另一个客户端对给定 `key` 的任意一个执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

```
redis> EXISTS job                # 确保两个 key 都不存在
(integer) 0
redis> EXISTS command
(integer) 0

redis> BLPOP job command 300     # 因为key一开始不存在，所以操作会被阻塞
1) "job"                         # 这里被 push 的是 job
2) "do my home work"             # 被弹出的值
(26.26s)                         # 等待的秒数

redis> BLPOP job command 5       # 等待超时的情况
(nil)
(5.66s)                          # 等待的秒数
```

相同的**key**被多个客户端同时阻塞

相同的 `key` 可以被多个客户端同时阻塞。

不同的客户端被放进一个队列中，按『先阻塞先服务』(first-BLPOP，first-served)的顺序为 `key` 执行 BLPOP 命令。

在**MULTI/EXEC**事务中的**BLPOP**

BLPOP 可以用于流水线(pipline,批量地发送多个命令并读入多个回复)，但把它用在 *MULTI* / *EXEC* 块当中没有意义。因为这要求整个服务器被阻塞以保证块执行时的原子性，该行为阻止了其他客户端执行 *LPUSH* 或 *RPUSH* 命令。

因此，一个被包裹在 *MULTI* / *EXEC* 块内的 BLPOP 命令，行为表现得就像 *LPOP* 一样，对空列表返回 `nil` ，对非空列表弹出列表元素，不进行任何阻塞操作。

```
# 对非空列表进行操作

redis> RPUSH job programming
(integer) 1

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC           # 不阻塞，立即返回
1) 1) "job"
   2) "programming"

# 对空列表进行操作

redis> LLEN job      # 空列表
(integer) 0

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC          # 不阻塞，立即返回
1) (nil)
```

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果列表为空，返回一个 `nil` 。否则，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

模式：事件提醒

有时候，为了等待一个新元素到达数据中，需要使用轮询的方式对数据进行探查。

另一种更好的方式是，使用系统提供的阻塞原语，在新元素到达时立即进行处理，而新元素还没到达时，就一直阻塞住，避免轮询占用资源。

对于 Redis ，我们似乎需要一个阻塞版的 *SPOP* 命令，但实际上，使用 BLPOP 或者 *BRPOP* 就能很好地解决这个问题。

使用元素的客户端(消费者)可以执行类似以下的代码：

```
LOOP forever
    WHILE SPOP(key) returns elements
        ... process elements ...
    END
    BRPOP helper_key
END
```

添加元素的客户端(消费者)则执行以下代码：

```
MULTI
    SADD key element
    LPUSH helper_key x
EXEC
```

BRPOP

BRPOP key [key ...] timeout

BRPOP 是列表的阻塞式(blocking)弹出原语。

它是 *RPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。

关于阻塞操作的更多信息，请查看 *BLPOP* 命令， BRPOP 除了弹出元素的位置和 *BLPOP* 不同之外，其他表现一致。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> BRPOP course 30
1) "course"             # 被弹出元素所属的列表键
2) "c++101"             # 被弹出的元素
```

BRPOPLPUSH

BRPOPLPUSH source destination timeout

BRPOPLPUSH 是 *RPOPLPUSH* 的阻塞版本，当给定列表 `source` 不为空时，BRPOPLPUSH 的表现和 *RPOPLPUSH* 一样。

当列表 `source` 为空时， BRPOPLPUSH 命令将阻塞连接，直到等待超时，或有另一个客户端对 `source` 执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

更多相关信息，请参考 *RPOPLPUSH* 命令。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长。

```
# 非空列表

redis> BRPOPLPUSH msg reciver 500
"hello moto"                        # 弹出元素的值
(3.38s)                             # 等待时长

redis> LLEN reciver
(integer) 1

redis> LRANGE reciver 0 0
1) "hello moto"

# 空列表

redis> BRPOPLPUSH msg reciver 1
(nil)
(1.34s)
```

模式：安全队列

参考 *RPOPLPUSH* 命令的『安全队列』模式。

模式：循环列表

参考 *RPOPLPUSH* 命令的『循环列表』模式。

LINDEX

LINDEX key index

返回列表 `key` 中，下标为 `index` 的元素。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为到达下标 `index` 过程中经过的元素数量。因此，对列表的头元素和尾元素执行 LINDEX 命令，复杂度为O(1)。

返回值:

列表中下标为 `index` 的元素。如果 `index` 参数的值不在列表的区间范围内(out of range)，返回 `nil` 。

```
redis> LPUSH mylist "World"
(integer) 1

redis> LPUSH mylist "Hello"
(integer) 2

redis> LINDEX mylist 0
"Hello"

redis> LINDEX mylist -1
"World"

redis> LINDEX mylist 3        # index不在 mylist 的区间范围内
(nil)
```

LINSERT

LINSERT key BEFORE|AFTER pivot value

将值 `value` 插入到列表 `key` 当中，位于值 `pivot` 之前或之后。

当 `pivot` 不存在于列表 `key` 时，不执行任何操作。

当 `key` 不存在时， `key` 被视为空列表，不执行任何操作。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 2.2.0

时间复杂度:

O(N)， `N` 为寻找 `pivot` 过程中经过的元素数量。

返回值:

如果命令执行成功，返回插入操作完成之后，列表的长度。如果没有找到 `pivot` ，返回 `-1` 。如果 `key` 不存在或为空列表，返回 `0` 。

```
redis> RPUSH mylist "Hello"
(integer) 1

redis> RPUSH mylist "World"
(integer) 2

redis> LINSERT mylist BEFORE "World" "There"
(integer) 3

redis> LRANGE mylist 0 -1
1) "Hello"
2) "There"
3) "World"

# 对一个非空列表插入，查找一个不存在的 pivot

redis> LINSERT mylist BEFORE "go" "let's"
(integer) -1                                    # 失败

# 对一个空列表执行 LINSERT 命令

redis> EXISTS fake_list
(integer) 0

redis> LINSERT fake_list BEFORE "nono" "gogogog"
(integer) 0                                      # 失败
```

LLEN

LLEN key

返回列表 `key` 的长度。

如果 `key` 不存在，则 `key` 被解释为一个空列表，返回 `0` .

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表 `key` 的长度。

```
# 空列表

redis> LLEN job
(integer) 0

# 非空列表

redis> LPUSH job "cook food"
(integer) 1

redis> LPUSH job "have lunch"
(integer) 2

redis> LLEN job
(integer) 2
```

LPOP

LPOP key

移除并返回列表 `key` 的头元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的头元素。当 `key` 不存在时，返回 `nil` 。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> LPOP course  # 移除头元素
"algorithm001"
```

LPUSH

LPUSH key value [value ...]

将一个或多个值 `value` 插入到列表 `key` 的表头

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表头： 比如说，对空列表 `mylist` 执行命令 `LPUSH mylist a b c` ，列表的值将是 `c b a` ，这等同于原子性地执行 `LPUSH mylist a` 、 `LPUSH mylist b` 和 `LPUSH mylist c` 三个命令。

如果 `key` 不存在，一个空列表会被创建并执行 LPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在Redis 2.4版本以前的 LPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 LPUSH 命令后，列表的长度。

```
# 加入单个元素

redis> LPUSH languages python
(integer) 1

# 加入重复元素

redis> LPUSH languages python
(integer) 2

redis> LRANGE languages 0 -1     # 列表允许重复元素
1) "python"
2) "python"

# 加入多个元素

redis> LPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "c"
2) "b"
3) "a"
```

LRANGE

LRANGE key start stop

返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

注意LRANGE命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表，对该列表执行 `LRANGE list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LRANGE 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` (`LLEN list` 减去 `1`)还要大，那么 LRANGE 返回一个空列表。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(S+N)， `S` 为偏移量 `start` ， `N` 为指定区间内元素的数量。

返回值:

一个列表，包含指定区间内的元素。

```
redis> RPUSH fp-language lisp
(integer) 1

redis> LRANGE fp-language 0 0
1) "lisp"

redis> RPUSH fp-language scheme
(integer) 2

redis> LRANGE fp-language 0 1
1) "lisp"
2) "scheme"
```

LREM

LREM key count value

根据参数 `count` 的值，移除列表中与参数 `value` 相等的元素。

`count` 的值可以是以下几种：

- `count > 0` : 从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count` 。
- `count < 0` : 从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。
- `count = 0` : 移除表中所有与 `value` 相等的值。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为列表的长度。

返回值：

被移除元素的数量。因为不存在的 `key` 被视作空表(empty list)，所以当 `key` 不存在时， LREM 命令总是返回 `0` 。

```
# 先创建一个表，内容排列是
# morning hello morning helllo morning

redis> LPUSH greet "morning"
(integer) 1
redis> LPUSH greet "hello"
(integer) 2
redis> LPUSH greet "morning"
(integer) 3
redis> LPUSH greet "hello"
(integer) 4
redis> LPUSH greet "morning"
(integer) 5

redis> LRANGE greet 0 4         # 查看所有元素
1) "morning"
2) "hello"
3) "morning"
4) "hello"
5) "morning"

redis> LREM greet 2 morning     # 移除从表头到表尾，最先发现的两个 morni
(integer) 2                     # 两个元素被移除

redis> LLEN greet               # 还剩 3 个元素
(integer) 3

redis> LRANGE greet 0 2
1) "hello"
2) "hello"
3) "morning"

redis> LREM greet -1 morning    # 移除从表尾到表头，第一个 morning
(integer) 1

redis> LLEN greet               # 剩下两个元素
(integer) 2

redis> LRANGE greet 0 1
1) "hello"
2) "hello"

redis> LREM greet 0 hello      # 移除表中所有 hello
(integer) 2                    # 两个 hello 被移除

redis> LLEN greet
(integer) 0
```

LSET

LSET key index value

将列表 `key` 下标为 `index` 的元素的值设置为 `value` 。

当 `index` 参数超出范围，或对一个空列表(`key` 不存在)进行 LSET 时，返回一个错误。

关于列表下标的更多信息，请参考 *LINDEX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

对头元素或尾元素进行 LSET 操作，复杂度为 O(1)。其他情况下，为 O(N)， `N` 为列表的长度。

返回值：

操作成功返回 `ok` ，否则返回错误信息。

```
# 对空列表(key 不存在)进行 LSET

redis> EXISTS list
(integer) 0

redis> LSET list 0 item
(error) ERR no such key

# 对非空列表进行 LSET

redis> LPUSH job "cook food"
(integer) 1

redis> LRANGE job 0 0
1) "cook food"

redis> LSET job 0 "play game"
OK

redis> LRANGE job  0 0
1) "play game"

# index 超出范围

redis> LLEN list                    # 列表长度为 1
(integer) 1

redis> LSET list 3 'out of range'
(error) ERR index out of range
```

LTRIM

LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

举个例子，执行命令 `LTRIM list 0 2` ，表示只保留列表 `list` 的前三个元素，其余元素全部删除。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

当 `key` 不是列表类型时，返回一个错误。

LTRIM 命令通常和 *LPUSH* 命令或 *RPUSH* 命令配合使用，举个例子：

```
LPUSH log newest_log
LTRIM log 0 99
```

这个例子模拟了一个日志程序，每次将最新日志 `newest_log` 放到 `log` 列表中，并且只保留最新的 `100` 项。注意当这样使用 `LTRIM` 命令时，时间复杂度是O(1)，因为平均情况下，每次只有一个元素被移除。

注意**LTRIM**命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表 `list` ，对该列表执行 `LTRIM list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LTRIM 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` (`LLEN list` 减去 `1`)还要大，或者 `start > stop` ， LTRIM 返回一个空列表(因为 LTRIM 已经将整个列表清空)。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为被移除的元素的数量。

返回值:

命令执行成功时，返回 `ok` 。

```
# 情况 1： 常见情况， start 和 stop 都在列表的索引范围之内

redis> LRANGE alpha 0 -1       # alpha 是一个包含 5 个字符串的列表
1) "h"
2) "e"
3) "l"
4) "l"
5) "o"

redis> LTRIM alpha 1 -1        # 删除 alpha 列表索引为 0 的元素
OK

redis> LRANGE alpha 0 -1       # "h" 被删除了
1) "e"
2) "l"
3) "l"
4) "o"

# 情况 2： stop 比列表的最大下标还要大

redis> LTRIM alpha 1 10086     # 保留 alpha 列表索引 1 至索引 10086 上
OK

redis> LRANGE alpha 0 -1       # 只有索引 0 上的元素 "e" 被删除了，其他
1) "l"
2) "l"
3) "o"

# 情况 3： start 和 stop 都比列表的最大下标要大，并且 start < stop

redis> LTRIM alpha 10086 123321
OK

redis> LRANGE alpha 0 -1        # 列表被清空
(empty list or set)

# 情况 4： start 和 stop 都比列表的最大下标要大，并且 start > stop

redis> RPUSH new-alpha "h" "e" "l" "l" "o"     # 重新建立一个新列表
(integer) 5

redis> LRANGE new-alpha 0 -1
1) "h"
2) "e"
3) "l"
```

```
4) "l"
5) "o"

redis> LTRIM new-alpha 123321 10086    # 执行 LTRIM
OK

redis> LRANGE new-alpha 0 -1           # 同样被清空
(empty list or set)
```

RPOP

RPOP key

移除并返回列表 `key` 的尾元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的尾元素。当 `key` 不存在时，返回 `nil` 。

```
redis> RPUSH mylist "one"
(integer) 1

redis> RPUSH mylist "two"
(integer) 2

redis> RPUSH mylist "three"
(integer) 3

redis> RPOP mylist           # 返回被弹出的元素
"three"

redis> LRANGE mylist 0 -1    # 列表剩下的元素
1) "one"
2) "two"
```

RPOPLPUSH

RPOPLPUSH source destination

命令 RPOPLPUSH 在一个原子时间内，执行以下两个动作：

- 将列表 `source` 中的最后一个元素(尾元素)弹出，并返回给客户端。
- 将 `source` 弹出的元素插入到列表 `destination` ，作为 `destination` 列表的的头元素。

举个例子，你有两个列表 `source` 和 `destination` ， `source` 列表有元素 `a, b, c` ， `destination` 列表有元素 `x, y, z` ，执行 `RPOPLPUSH source destination` 之后， `source` 列表包含元素 `a, b` ， `destination` 列表包含元素 `c, x, y, z` ，并且元素 `c` 会被返回给客户端。

如果 `source` 不存在，值 `nil` 被返回，并且不执行其他动作。

如果 `source` 和 `destination` 相同，则列表中的表尾元素被移动到表头，并返回该元素，可以把这种特殊情况视作列表的旋转(rotation)操作。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

被弹出的元素。

```
# source 和 destination 不同

redis> LRANGE alpha 0 -1         # 查看所有元素
1) "a"
2) "b"
3) "c"
4) "d"

redis> RPOPLPUSH alpha reciver   # 执行一次 RPOPLPUSH 看看
"d"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"
3) "c"

redis> LRANGE reciver 0 -1
1) "d"
```

```
redis> RPOPLPUSH alpha reciver   # 再执行一次，证实 RPOP 和 LPUSH 的位
"c"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"

redis> LRANGE reciver 0 -1
1) "c"
2) "d"

# source 和 destination 相同

redis> LRANGE number 0 -1
1) "1"
2) "2"
3) "3"
4) "4"

redis> RPOPLPUSH number number
"4"

redis> LRANGE number 0 -1          # 4 被旋转到了表头
1) "4"
2) "1"
3) "2"
4) "3"

redis> RPOPLPUSH number number
"3"

redis> LRANGE number 0 -1          # 这次是 3 被旋转到了表头
1) "3"
2) "4"
3) "1"
4) "2"
```

模式： 安全的队列

Redis的列表经常被用作队列(queue)，用于在不同程序之间有序地交换消息(message)。一个客户端通过 *LPUSH* 命令将消息放入队列中，而另一个客户端通过 *RPOP* 或者 *BRPOP* 命令取出队列中等待时间最长的消息。

不幸的是，上面的队列方法是『不安全』的，因为在这个过程中，一个客户端可能在取出一个消息之后崩溃，而未处理完的消息也就因此丢失。

使用 RPOPLPUSH 命令(或者它的阻塞版本 *BRPOPLPUSH*)可以解决这个问题：因为它不仅返回一个消息，同时还将这个消息添加到另一个备份列表当中，如果一切正常的话，当一个客户端完成某个消息的处理之后，可以用 *LREM* 命令将这个消息从备份表删除。

最后，还可以添加一个客户端专门用于监视备份表，它自动地将超过一定处理时限的消息重新放入队列中去(负责处理该消息的客户端可能已经崩溃)，这样就不会丢失任何消息了。

模式：循环列表

通过使用相同的 `key` 作为 RPOPLPUSH 命令的两个参数，客户端可以用一个接一个地获取列表元素的方式，取得列表的所有元素，而不必像 *LRANGE* 命令那样一下子将所有列表元素都从服务器传送到客户端中(两种方式的总复杂度都是O(N))。

以上的模式甚至在以下的两个情况下也能正常工作：

- 有多个客户端同时对同一个列表进行旋转(rotating)，它们获取不同的元素，直到所有元素都被读取完，之后又从头开始。
- 有客户端在向列表尾部(右边)添加新元素。

这个模式使得我们可以很容易实现这样一类系统：有 N 个客户端，需要连续不断地对一些元素进行处理，而且处理的过程必须尽可能地快。一个典型的例子就是服务器的监控程序：它们需要在尽可能短的时间内，并行地检查一组网站，确保它们的可访问性。

注意，使用这个模式的客户端是易于扩展(scala)且安全(reliable)的，因为就算接收到元素的客户端失败，元素还是保存在列表里面，不会丢失，等到下个迭代来临的时候，别的客户端又可以继续处理这些元素了。

RPUSH

RPUSH key value [value ...]

将一个或多个值 `value` 插入到列表 `key` 的表尾(最右边)。

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表尾：比如对一个空列表 `mylist` 执行 `RPUSH mylist a b c` ，得出的结果列表为 `a b c` ，等同于执行命令 `RPUSH mylist a` 、 `RPUSH mylist b` 、 `RPUSH mylist c` 。

如果 `key` 不存在，一个空列表会被创建并执行 RPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在 Redis 2.4 版本以前的 RPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 RPUSH 操作后，表的长度。

```
# 添加单个元素

redis> RPUSH languages c
(integer) 1

# 添加重复元素

redis> RPUSH languages c
(integer) 2

redis> LRANGE languages 0 -1 # 列表允许重复元素
1) "c"
2) "c"

# 添加多个元素

redis> RPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "a"
2) "b"
3) "c"
```

RPUSHX

RPUSHX key value

将值 `value` 插入到列表 `key` 的表尾，当且仅当 `key` 存在并且是一个列表。

和 *RPUSH* 命令相反，当 `key` 不存在时， RPUSHX 命令什么也不做。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

RPUSHX 命令执行之后，表的长度。

```
# key不存在

redis> LLEN greet
(integer) 0

redis> RPUSHX greet "hello"     # 对不存在的 key 进行 RPUSHX，PUSH 失
(integer) 0

# key 存在且是一个非空列表

redis> RPUSH greet "hi"         # 先用 RPUSH 插入一个元素
(integer) 1

redis> RPUSHX greet "hello"     # greet 现在是一个列表类型，RPUSHX 操作
(integer) 2

redis> LRANGE greet 0 -1
1) "hi"
2) "hello"
```

Set（集合）

SADD

SADD key member [member ...]

将一个或多个 `member` 元素加入到集合 `key` 当中，已经存在于集合的 `member` 元素将被忽略。

假如 `key` 不存在，则创建一个只包含 `member` 元素作成员的集合。

当 `key` 不是集合类型时，返回一个错误。

Note

在Redis2.4版本以前， SADD 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是被添加的元素的数量。

返回值:

被添加到集合中的新元素的数量，不包括被忽略的元素。

```
# 添加单个元素

redis> SADD bbs "discuz.net"
(integer) 1

# 添加重复元素

redis> SADD bbs "discuz.net"
(integer) 0

# 添加多个元素

redis> SADD bbs "tianya.cn" "groups.google.com"
(integer) 2

redis> SMEMBERS bbs
1) "discuz.net"
2) "groups.google.com"
3) "tianya.cn"
```

SCARD

SCARD key

返回集合 `key` 的基数(集合中元素的数量)。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值：

集合的基数。当 `key` 不存在时，返回 `0` 。

```
redis> SADD tool pc printer phone
(integer) 3

redis> SCARD tool   # 非空集合
(integer) 3

redis> DEL tool
(integer) 1

redis> SCARD tool   # 空集合
(integer) 0
```

SDIFF

SDIFF key [key ...]

返回一个集合的全部成员，该集合是所有给定集合之间的差集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

一个包含差集成员的列表。

```
redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SDIFF peter's_movies joe's_movies
1) "bet man"
2) "start war"
```

SDIFFSTORE

SDIFFSTORE destination key [key ...]

这个命令的作用和 *SDIFF* 类似，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SDIFFSTORE joe_diff_peter joe's_movies peter's_movies
(integer) 2

redis> SMEMBERS joe_diff_peter
1) "hi, lady"
2) "Fast Five"
```

SINTER

SINTER key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

SINTER

SINTER key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

SINTERSTORE

SINTERSTORE destination key [key ...]

这个命令类似于 *SINTER* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

结果集中的成员数量。

```
redis> SMEMBERS songs
1) "good bye joe"
2) "hello,peter"

redis> SMEMBERS my_songs
1) "good bye joe"
2) "falling"

redis> SINTERSTORE song_interset songs my_songs
(integer) 1

redis> SMEMBERS song_interset
1) "good bye joe"
```

SISMEMBER

SISMEMBER key member

判断 `member` 元素是否集合 `key` 的成员。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素是集合的成员，返回 `1` 。如果 `member` 元素不是集合的成员，或 `key` 不存在，返回 `0` 。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SISMEMBER joe's_movies "bet man"
(integer) 0

redis> SISMEMBER joe's_movies "Fast Five"
(integer) 1
```

SMEMBERS

SMEMBERS key

返回集合 `key` 中的所有成员。

不存在的 `key` 被视为空集合。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为集合的基数。

返回值:

集合中的所有成员。

```
# key 不存在或集合为空

redis> EXISTS not_exists_key
(integer) 0

redis> SMEMBERS not_exists_key
(empty list or set)

# 非空集合

redis> SADD language Ruby Python Clojure
(integer) 3

redis> SMEMBERS language
1) "Python"
2) "Ruby"
3) "Clojure"
```

SMOVE

SMOVE source destination member

将 `member` 元素从 `source` 集合移动到 `destination` 集合。

SMOVE 是原子性操作。

如果 `source` 集合不存在或不包含指定的 `member` 元素，则 SMOVE 命令不执行任何操作，仅返回 `0` 。否则， `member` 元素从 `source` 集合中被移除，并添加到 `destination` 集合中去。

当 `destination` 集合已经包含 `member` 元素时， SMOVE 命令只是简单地将 `source` 集合中的 `member` 元素删除。

当 `source` 或 `destination` 不是集合类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素被成功移除，返回 `1` 。如果 `member` 元素不是 `source` 集合的成员，并且没有任何操作对 `destination` 集合执行，那么返回 `0` 。

```
redis> SMEMBERS songs
1) "Billie Jean"
2) "Believe Me"

redis> SMEMBERS my_songs
(empty list or set)

redis> SMOVE songs my_songs "Believe Me"
(integer) 1

redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"
```

SPOP

SPOP key

移除并返回集合中的一个随机元素。

如果只想获取一个随机元素，但不想该元素从集合中被移除的话，可以使用 *SRANDMEMBER* 命令。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

被移除的随机元素。当 `key` 不存在或 `key` 是空集时，返回 `nil` 。

```
redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"
3) "Redis"

redis> SPOP db
"Redis"

redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"

redis> SPOP db
"MySQL"

redis> SMEMBERS db
1) "MongoDB"
```

SRANDMEMBER

SRANDMEMBER key [count]

如果命令执行时，只提供了 `key` 参数，那么返回集合中的一个随机元素。

从 Redis 2.6 版本开始， SRANDMEMBER 命令接受可选的 `count` 参数：

- 如果 `count` 为正数，且小于集合基数，那么命令返回一个包含 `count` 个元素的数组，数组中的元素各不相同。如果 `count` 大于等于集合基数，那么返回整个集合。
- 如果 `count` 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 `count` 的绝对值。

该操作和 *SPOP* 相似，但 *SPOP* 将随机元素从集合中移除并返回，而 SRANDMEMBER 则仅仅返回随机元素，而不对集合进行任何改动。

可用版本：

>= 1.0.0

时间复杂度:

只提供 `key` 参数时为 O(1) 。如果提供了 `count` 参数，那么为 O(N) ，N 为返回数组的元素个数。

返回值:

只提供 `key` 参数时，返回一个元素；如果集合为空，返回 `nil` 。如果提供了 `count` 参数，那么返回一个数组；如果集合为空，返回空数组。

```
# 添加元素

redis> SADD fruit apple banana cherry
(integer) 3

# 只给定 key 参数，返回一个随机元素

redis> SRANDMEMBER fruit
"cherry"

redis> SRANDMEMBER fruit
"apple"

# 给定 3 为 count 参数，返回 3 个随机元素
# 每个随机元素都不相同

redis> SRANDMEMBER fruit 3
1) "apple"
```

```
2) "banana"
3) "cherry"

# 给定 -3 为 count 参数，返回 3 个随机元素
# 元素可能会重复出现多次

redis> SRANDMEMBER fruit -3
1) "banana"
2) "cherry"
3) "apple"

redis> SRANDMEMBER fruit -3
1) "apple"
2) "apple"
3) "cherry"

# 如果 count 是整数，且大于等于集合基数，那么返回整个集合

redis> SRANDMEMBER fruit 10
1) "apple"
2) "banana"
3) "cherry"

# 如果 count 是负数，且 count 的绝对值大于集合的基数
# 那么返回的数组的长度为 count 的绝对值

redis> SRANDMEMBER fruit -10
1) "banana"
2) "apple"
3) "banana"
4) "cherry"
5) "apple"
6) "apple"
7) "cherry"
8) "apple"
9) "apple"
10) "banana"

# SRANDMEMBER 并不会修改集合内容

redis> SMEMBERS fruit
1) "apple"
2) "cherry"
3) "banana"

# 集合为空时返回 nil 或者空数组

redis> SRANDMEMBER not-exists
(nil)

redis> SRANDMEMBER not-eixsts 10
(empty list or set)
```

SREM

SREM key member [member ...]

移除集合 `key` 中的一个或多个 `member` 元素，不存在的 `member` 元素会被忽略。

当 `key` 不是集合类型，返回一个错误。

Note

在 Redis 2.4 版本以前， SREM 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为给定 `member` 元素的数量。

返回值:

被成功移除的元素的数量，不包括被忽略的元素。

```
# 测试数据

redis> SMEMBERS languages
1) "c"
2) "lisp"
3) "python"
4) "ruby"

# 移除单个元素

redis> SREM languages ruby
(integer) 1

# 移除不存在元素

redis> SREM languages non-exists-language
(integer) 0

# 移除多个元素

redis> SREM languages lisp python c
(integer) 3

redis> SMEMBERS languages
(empty list or set)
```

SUNION

SUNION key [key ...]

返回一个集合的全部成员，该集合是所有给定集合的并集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

并集成员的列表。

```
redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"

redis> SUNION songs my_songs
1) "Billie Jean"
2) "Believe Me"
```

SUNIONSTORE

SUNIONSTORE destination key [key ...]

这个命令类似于 *SUNION* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS NoSQL
1) "MongoDB"
2) "Redis"

redis> SMEMBERS SQL
1) "sqlite"
2) "MySQL"

redis> SUNIONSTORE db NoSQL SQL
(integer) 4

redis> SMEMBERS db
1) "MySQL"
2) "sqlite"
3) "MongoDB"
4) "Redis"
```

SSCAN

SSCAN key cursor [MATCH pattern] [COUNT count]

详细信息请参考 *SCAN* 命令。

SortedSet（有序集合）

ZADD

ZADD key score member [[score member] [score member] ...]

将一个或多个 `member` 元素及其 `score` 值加入到有序集 `key` 当中。

如果某个 `member` 已经是有序集的成员，那么更新这个 `member` 的 `score` 值，并通过重新插入这个 `member` 元素，来保证该 `member` 在正确的位置上。

`score` 值可以是整数值或双精度浮点数。

如果 `key` 不存在，则创建一个空的有序集并执行 ZADD 操作。

当 `key` 存在但不是有序集类型时，返回一个错误。

对有序集的更多介绍请参见 sorted set 。

Note

在 Redis 2.4 版本以前， ZADD 每次只能添加一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 是有序集的基数， `M` 为成功添加的新成员的数量。

返回值:

被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。

```
# 添加单个元素

redis> ZADD page_rank 10 google.com
(integer) 1

# 添加多个元素

redis> ZADD page_rank 9 baidu.com 8 bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，且 score 值不变

redis> ZADD page_rank 10 google.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # 没有改变
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，但是改变 score 值

redis> ZADD page_rank 6 bing.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # bing.com 元素的 score 值衤
1) "bing.com"
2) "6"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"
```

ZCARD

ZCARD key

返回有序集 `key` 的基数。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

当 `key` 存在且是有序集类型时，返回有序集的基数。当 `key` 不存在时，返回 `0` 。

```
redis > ZADD salary 2000 tom    # 添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 1

redis > ZADD salary 5000 jack   # 再添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 2

redis > EXISTS non_exists_key   # 对不存在的 key 进行 ZCARD 操作
(integer) 0

redis > ZCARD non_exists_key
(integer) 0
```

ZCOUNT

ZCOUNT key min max

返回有序集 `key` 中， `score` 值在 `min` 和 `max` 之间(默认包括 `score` 值等于 `min` 或 `max`)的成员的数量。

关于参数 `min` 和 `max` 的详细使用方法，请参考 *ZRANGEBYSCORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))， `N` 为有序集的基数。

返回值:

`score` 值在 `min` 和 `max` 之间的成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZCOUNT salary 2000 5000          # 计算薪水在 2000-5000 之间的
(integer) 3

redis> ZCOUNT salary 3000 5000          # 计算薪水在 3000-5000 之间的
(integer) 2
```

ZINCRBY

ZINCRBY key increment member

为有序集 `key` 的成员 `member` 的 `score` 值加上增量 `increment` 。

可以通过传递一个负数值 `increment` ，让 `score` 减去相应的值，比如 `ZINCRBY key -5 member` ，就是让 `member` 的 `score` 值减去 `5` 。

当 `key` 不存在，或 `member` 不是 `key` 的成员时， `ZINCRBY key increment member` 等同于 `ZADD key increment member` 。

当 `key` 不是有序集类型时，返回一个错误。

`score` 值可以是整数值或双精度浮点数。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N))

返回值:

`member` 成员的新 `score` 值，以字符串形式表示。

```
redis> ZSCORE salary tom
"2000"

redis> ZINCRBY salary 2000 tom   # tom 加薪啦！
"4000"
```

ZRANGE

ZRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递增(从小到大)来排序。

具有相同 `score` 值的成员按字典序(lexicographical order)来排列。

如果你需要成员按 `score` 值递减(从大到小)来排列，请使用 *ZREVRANGE* 命令。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。超出范围的下标并不会引起错误。比如说，当 `start` 的值比有序集的最大下标还要大，或是 `start > stop` 时， ZRANGE 命令只是简单地返回一个空列表。另一方面，假如 `stop` 参数的值比有序集的最大下标还要大，那么 Redis 将 `stop` 当作最大下标来处理。可以通过使用 `WITHSCORES` 选项，来让成员和它的 `score` 值一并返回，返回列表以 `value1,score1, ..., valueN,scoreN` 的格式表示。客户端库可能会返回一些更复杂的数据类型，比如数组、元组等。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZRANGE salary 0 -1 WITHSCORES             # 显示整个有序集成
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 1 2 WITHSCORES              # 显示有序集下标区
1) "tom"
2) "5000"
3) "boss"
4) "10086"

redis > ZRANGE salary 0 200000 WITHSCORES         # 测试 end 下标超出
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 200000 3000000 WITHSCORES   # 测试当给定区间不存
(empty list or set)
```

ZRANGEBYSCORE

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

返回有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员。有序集成员按 `score` 值递增(从小到大)次序排列。

具有相同 `score` 值的成员按字典序(lexicographical order)来排列(该属性是有序集提供的，不需要额外的计算)。

可选的 `LIMIT` 参数指定返回结果的数量及区间(就像SQL中的 `SELECT LIMIT offset, count`)，注意当 `offset` 很大时，定位 `offset` 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间。

可选的 `WITHSCORES` 参数决定结果集是单单返回有序集的成员，还是将有序集成员及其 `score` 值一起返回。该选项自 Redis 2.0 版本起可用。

区间及无限

`min` 和 `max` 可以是 `-inf` 和 `+inf` ，这样一来，你就可以在不知道有序集的最低和最高 `score` 值的情况下，使用 ZRANGEBYSCORE 这类命令。

默认情况下，区间的取值使用闭区间 (小于等于或大于等于)，你也可以通过给参数前增加 `(` 符号来使用可选的开区间 (小于或大于)。

举个例子：

```
ZRANGEBYSCORE zset (1 5
```

返回所有符合条件 `1 < score <= 5` 的成员，而

```
ZRANGEBYSCORE zset (5 (10
```

则返回所有符合条件 `5 < score < 10` 的成员。

可用版本：

>= 1.0.5

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为被结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZADD salary 2500 jack                        # 测试数据
(integer) 0
redis> ZADD salary 5000 tom
(integer) 0
redis> ZADD salary 12000 peter
(integer) 0

redis> ZRANGEBYSCORE salary -inf +inf               # 显示整个有序集
1) "jack"
2) "tom"
3) "peter"

redis> ZRANGEBYSCORE salary -inf +inf WITHSCORES    # 显示整个有序集及
1) "jack"
2) "2500"
3) "tom"
4) "5000"
5) "peter"
6) "12000"

redis> ZRANGEBYSCORE salary -inf 5000 WITHSCORES    # 显示工资 <=500
1) "jack"
2) "2500"
3) "tom"
4) "5000"

redis> ZRANGEBYSCORE salary (5000 400000            # 显示工资大于 50
1) "peter"
```

ZRANK

ZRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递增(从小到大)顺序排列。

排名以 `0` 为底，也就是说， `score` 值最小的成员排名为 `0` 。

使用 *ZREVRANK* 命令可以获得成员按 `score` 值递减(从大到小)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 显示所有成员及其 score
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZRANK salary tom                     # 显示 tom 的薪水排名，第
(integer) 1
```

ZREM

ZREM key member [member ...]

移除有序集 `key` 中的一个或多个成员，不存在的成员将被忽略。

当 `key` 存在但不是有序集类型时，返回一个错误。

Note

在 Redis 2.4 版本以前， ZREM 每次只能删除一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 为有序集的基数， `M` 为被成功移除的成员的数量。

返回值:

被成功移除的成员的数量，不包括被忽略的成员。

```
# 测试数据

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 移除单个元素

redis> ZREM page_rank google.com
(integer) 1

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"

# 移除多个元素

redis> ZREM page_rank baidu.com bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
(empty list or set)

# 移除不存在元素

redis> ZREM page_rank non-exists-element
(integer) 0
```

ZREMRANGEBYRANK

ZREMRANGEBYRANK key start stop

移除有序集 `key` 中，指定排名(rank)区间内的所有成员。

区间分别以下标参数 `start` 和 `stop` 指出，包含 `start` 和 `stop` 在内。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZADD salary 2000 jack
(integer) 1
redis> ZADD salary 5000 tom
(integer) 1
redis> ZADD salary 3500 peter
(integer) 1

redis> ZREMRANGEBYRANK salary 0 1       # 移除下标 0 至 1 区间内的成员
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES    # 有序集只剩下一个成员
1) "tom"
2) "5000"
```

ZREMRANGEBYSCORE

ZREMRANGEBYSCORE key min max

移除有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max`)的成员。

自版本2.1.6开始， `score` 值等于 `min` 或 `max` 的成员也可以不包括在内，详情请参见 *ZRANGEBYSCORE* 命令。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES          # 显示有序集内所有成员及
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZREMRANGEBYSCORE salary 1500 3500      # 移除所有薪水在 1500
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES          # 剩下的有序集成员
1) "jack"
2) "5000"
```

ZREVRANGE

ZREVRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递减(从大到小)来排列。具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGE 命令的其他方面和 *ZRANGE* 命令一样。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 递增排列
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZREVRANGE salary 0 -1 WITHSCORES     # 递减排列
1) "jack"
2) "5000"
3) "tom"
4) "4000"
5) "peter"
6) "3500"
```

ZREVRANGEBYSCORE

ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

返回有序集 `key` 中， `score` 值介于 `max` 和 `min` 之间(默认包括等于 `max` 或 `min`)的所有的成员。有序集成员按 `score` 值递减(从大到小)的次序排列。

具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGEBYSCORE 命令的其他方面和 *ZRANGEBYSCORE* 命令一样。

可用版本：

>= 2.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZADD salary 10086 jack
(integer) 1
redis > ZADD salary 5000 tom
(integer) 1
redis > ZADD salary 7500 peter
(integer) 1
redis > ZADD salary 3500 joe
(integer) 1

redis > ZREVRANGEBYSCORE salary +inf -inf   # 逆序排列所有成员
1) "jack"
2) "peter"
3) "tom"
4) "joe"

redis > ZREVRANGEBYSCORE salary 10000 2000  # 逆序排列薪水介于 10000
1) "peter"
2) "tom"
3) "joe"
```

ZREVRANK

ZREVRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递减(从大到小)排序。

排名以 `0` 为底，也就是说， `score` 值最大的成员排名为 `0` 。

使用 *ZRANK* 命令可以获得成员按 `score` 值递增(从小到大)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis 127.0.0.1:6379> ZRANGE salary 0 -1 WITHSCORES     # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZREVRANK salary peter     # peter 的工资排第二
(integer) 1

redis> ZREVRANK salary tom       # tom 的工资最高
(integer) 0
```

ZSCORE

ZSCORE key member

返回有序集 `key` 中，成员 `member` 的 `score` 值。

如果 `member` 元素不是有序集 `key` 的成员，或 `key` 不存在，返回 `nil` 。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

`member` 成员的 `score` 值，以字符串形式表示。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZSCORE salary peter              # 注意返回值是字符串
"3500"
```

ZUNIONSTORE

ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的并集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该并集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之 和 。

WEIGHTS

使用 `WEIGHTS` 选项，你可以为 每个 给定有序集 分别 指定一个乘法因子(multiplication factor)，每个给定有序集的所有成员的 `score` 值在传递给聚合函数(aggregation function)之前都要先乘以该有序集的因子。

如果没有指定 `WEIGHTS` 选项，乘法因子默认设置为 `1` 。

AGGREGATE

使用 `AGGREGATE` 选项，你可以指定并集的结果集的聚合方式。

默认使用的参数 `SUM` ，可以将所有集合中某个成员的 `score` 值之 和 作为结果集中该成员的 `score` 值；使用参数 `MIN` ，可以将所有集合中某个成员的 最小 `score` 值作为结果集中该成员的 `score` 值；而参数 `MAX` 则是将所有集合中某个成员的 最大 `score` 值作为结果集中该成员的 `score` 值。

可用版本：

>= 2.0.0

时间复杂度:

O(N)+O(M log(M))， `N` 为给定有序集基数的总和， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis> ZRANGE programmer 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"

redis> ZRANGE manager 0 -1 WITHSCORES
1) "herry"
2) "2000"
3) "mary"
4) "3500"
5) "bob"
6) "4000"

redis> ZUNIONSTORE salary 2 programmer manager WEIGHTS 1 3   # 公司
(integer) 6

redis> ZRANGE salary 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"
7) "herry"
8) "6000"
9) "mary"
10) "10500"
11) "bob"
12) "12000"
```

ZINTERSTORE

ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的交集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该交集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之和.

关于 `WEIGHTS` 和 `AGGREGATE` 选项的描述，参见 *ZUNIONSTORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(N*K)+O(M*log(M))， `N` 为给定 `key` 中基数最小的有序集， `K` 为给定有序集的数量， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis > ZADD mid_test 70 "Li Lei"
(integer) 1
redis > ZADD mid_test 70 "Han Meimei"
(integer) 1
redis > ZADD mid_test 99.5 "Tom"
(integer) 1

redis > ZADD fin_test 88 "Li Lei"
(integer) 1
redis > ZADD fin_test 75 "Han Meimei"
(integer) 1
redis > ZADD fin_test 99.5 "Tom"
(integer) 1

redis > ZINTERSTORE sum_point 2 mid_test fin_test
(integer) 3

redis > ZRANGE sum_point 0 -1 WITHSCORES     # 显示有序集内所有成员及其
1) "Han Meimei"
2) "145"
3) "Li Lei"
4) "158"
5) "Tom"
6) "199"
```

ZSCAN

ZSCAN key cursor [MATCH pattern] [COUNT count]

详细信息请参考 *SCAN* 命令。

Pub/Sub（发布/订阅）

PSUBSCRIBE

PSUBSCRIBE pattern [pattern ...]

订阅一个或多个符合给定模式的频道。

每个模式以 `*` 作为匹配符，比如 `it*` 匹配所有以 `it` 开头的频道(`it.news` 、 `it.blog` 、 `it.tweets` 等等)， `news.*` 匹配所有以 `news.` 开头的频道(`news.it` 、 `news.global.today` 等等)，诸如此类。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是订阅的模式的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 news.* 和 tweet.* 两个模式

# 第 1 - 6 行是执行 psubscribe 之后的反馈信息
# 第 7 - 10 才是接收到的第一条信息
# 第 11 - 14 是第二条
# 以此类推。。。

redis> psubscribe news.* tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"                    # 返回值的类型：显示订阅成功
2) "news.*"                        # 订阅的模式
3) (integer) 1                     # 目前已订阅的模式的数量

1) "psubscribe"
2) "tweet.*"
3) (integer) 2

1) "pmessage"                      # 返回值的类型：信息
2) "news.*"                        # 信息匹配的模式
3) "news.it"                       # 信息本身的目标频道
4) "Google buy Motorola"           # 信息的内容

1) "pmessage"
2) "tweet.*"
3) "tweet.huangz"
4) "hello"

1) "pmessage"
2) "tweet.*"
3) "tweet.joe"
4) "@huangz morning"

1) "pmessage"
2) "news.*"
3) "news.life"
4) "An apple a day, keep doctors away"
```

PUBLISH

PUBLISH channel message

将信息 `message` 发送到指定的频道 `channel` 。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是频道 `channel` 的订阅者数量，而 `M` 则是使用模式订阅(subscribed patterns)的客户端的数量。

返回值：

接收到信息 `message` 的订阅者数量。

```
# 对没有订阅者的频道发送信息

redis> publish bad_channel "can any body hear me?"
(integer) 0

# 向有一个订阅者的频道发送信息

redis> publish msg "good morning"
(integer) 1

# 向有多个订阅者的频道发送信息

redis> publish chat_room "hello~ everyone"
(integer) 3
```

PUBSUB

PUBSUB <subcommand> [argument [argument ...]]

PUBSUB 是一个查看订阅与发布系统状态的内省命令， 它由数个不同格式的子命令组成， 以下将分别对这些子命令进行介绍。

可用版本： >= 2.8.0

PUBSUB CHANNELS [pattern]

列出当前的活跃频道。

活跃频道指的是那些至少有一个订阅者的频道， 订阅模式的客户端不计算在内。

`pattern` 参数是可选的：

- 如果不给出 `pattern` 参数，那么列出订阅与发布系统中的所有活跃频道。
- 如果给出 `pattern` 参数，那么只列出和给定模式 `pattern` 相匹配的那些活跃频道。

复杂度： O(N) ， `N` 为活跃频道的数量（对于长度较短的频道和模式来说，将进行模式匹配的复杂度视为常数）。

返回值： 一个由活跃频道组成的列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# 首先， client-3 打印所有活跃频道
# 注意，即使一个频道有多个订阅者，它也只输出一次，比如 news.it

client-3> PUBSUB CHANNELS
1) "news.sport"
2) "news.internet"
3) "news.it"

# 接下来， client-3 打印那些与模式 news.i* 相匹配的活跃频道
# 因为 news.sport 不匹配 news.i* ，所以它没有被打印

redis> PUBSUB CHANNELS news.i*
1) "news.internet"
2) "news.it"
```

PUBSUB NUMSUB [channel-1 ... channel-N]

返回给定频道的订阅者数量， 订阅模式的客户端不计算在内。

复杂度： O(N) ， `N` 为给定频道的数量。

返回值： 一个多条批量回复（Multi-bulk reply）， 回复中包含给定的频道，以及频道的订阅者数量。 格式为：频道 `channel-1` ， `channel-1` 的订阅者数量，频道 `channel-2` ， `channel-2` 的订阅者数量，诸如此类。 回复中频道的排列顺序和执行命令时给定频道的排列顺序一致。 不给定任何频道而直接调用这个命令也是可以的， 在这种情况下， 命令只返回一个空列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# client-3 打印各个频道的订阅者数量

client-3> PUBSUB NUMSUB news.it news.internet news.sport news.musi
1) "news.it"    # 频道
2) "2"          # 订阅该频道的客户端数量
3) "news.internet"
4) "1"
5) "news.sport"
6) "1"
7) "news.music" # 没有任何订阅者
8) "0"
```

PUBSUB NUMPAT

返回订阅模式的数量。

注意， 这个命令返回的不是订阅模式的客户端的数量， 而是客户端订阅的所有模式的数量总和。

复杂度： O(1) 。

返回值： 一个整数回复（Integer reply）。

```
# client-1 订阅 news.* 和 discount.* 两个模式

client-1> PSUBSCRIBE news.* discount.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1
1) "psubscribe"
2) "discount.*"
3) (integer) 2

# client-2 订阅 tweet.* 一个模式

client-2> PSUBSCRIBE tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "tweet.*"
3) (integer) 1

# client-3 返回当前订阅模式的数量为 3

client-3> PUBSUB NUMPAT
(integer) 3

# 注意，当有多个客户端订阅相同的模式时，相同的订阅也被计算在 PUBSUB NUMPAT 之
# 比如说，再新建一个客户端 client-4 ，让它也订阅 news.* 频道

client-4> PSUBSCRIBE news.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1

# 这时再计算被订阅模式的数量，就会得到数量为 4

client-3> PUBSUB NUMPAT
(integer) 4
```

PUNSUBSCRIBE

PUNSUBSCRIBE [pattern [pattern ...]]

指示客户端退订所有给定模式。

如果没有模式被指定，也即是，一个无参数的 `PUNSUBSCRIBE` 调用被执行，那么客户端使用 *PSUBSCRIBE* 命令订阅的所有模式都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的模式。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是客户端已订阅的模式的数量， `M` 则是系统中所有客户端订阅的模式的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

SUBSCRIBE

SUBSCRIBE channel [channel ...]

订阅给定的一个或多个频道的信息。

可用版本：

>= 2.0.0

时间复杂度：

O(N)，其中 `N` 是订阅的频道的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 msg 和 chat_room 两个频道

# 1 - 6 行是执行 subscribe 之后的反馈信息
# 第 7 - 9 行才是接收到的第一条信息
# 第 10 - 12 行是第二条

redis> subscribe msg chat_room
Reading messages... (press Ctrl-C to quit)
1) "subscribe"       # 返回值的类型：显示订阅成功
2) "msg"             # 订阅的频道名字
3) (integer) 1       # 目前已订阅的频道数量

1) "subscribe"
2) "chat_room"
3) (integer) 2

1) "message"         # 返回值的类型：信息
2) "msg"             # 来源(从那个频道发送过来)
3) "hello moto"      # 信息内容

1) "message"
2) "chat_room"
3) "testing...haha"
```

UNSUBSCRIBE

UNSUBSCRIBE [channel [channel ...]]

指示客户端退订给定的频道。

如果没有频道被指定，也即是，一个无参数的 `UNSUBSCRIBE` 调用被执行，那么客户端使用 *SUBSCRIBE* 命令订阅的所有频道都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的频道。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是客户端已订阅的频道的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

Transaction（事务）

DISCARD

DISCARD

取消事务，放弃执行事务块内的所有命令。

如果正在使用 *WATCH* 命令监视某个(或某些) key，那么取消所有监视，等同于执行命令 *UNWATCH* 。

可用版本：

>= 2.0.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI
OK

redis> PING
QUEUED

redis> SET greeting "hello"
QUEUED

redis> DISCARD
OK
```

EXEC

EXEC

执行所有事务块内的命令。

假如某个(或某些) key 正处于 *WATCH* 命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么 EXEC 命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。

可用版本：

>= 1.2.0

时间复杂度：

事务块内所有命令的时间复杂度的总和。

返回值：

事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 `nil` 。

```
# 事务被成功执行

redis> MULTI
OK

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG

# 监视 key ，且事务成功执行

redis> WATCH lock lock_times
OK
```

```
redis> MULTI
OK

redis> SET lock "huangz"
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC
1) OK
2) (integer) 1

# 监视 key ，且事务被打断

redis> WATCH lock lock_times
OK

redis> MULTI
OK

redis> SET lock "joe"        # 就在这时，另一个客户端修改了 lock_times
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC                  # 因为 lock_times 被修改， joe 的事务执行
(nil)
```

MULTI

MULTI

标记一个事务块的开始。

事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 *EXEC* 命令原子性(atomic)地执行。

可用版本：

>= 1.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI                # 标记事务开始
OK

redis> INCR user_id         # 多条命令按顺序入队
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC                 # 执行
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG
```

UNWATCH

UNWATCH

取消 *WATCH* 命令对所有 key 的监视。

如果在执行 *WATCH* 命令之后， *EXEC* 命令或 *DISCARD* 命令先被执行了的话，那么就不需要再执行 UNWATCH 了。

因为 *EXEC* 命令会执行事务，因此 *WATCH* 命令的效果已经产生了；而 *DISCARD* 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 UNWATCH 了。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

总是 `OK` 。

```
redis> WATCH lock lock_times
OK

redis> UNWATCH
OK
```

WATCH

WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

可用版本：

>= 2.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> WATCH lock lock_times
OK
```

Script（脚本）

EVAL

EVAL script numkeys key [key ...] arg [arg ...]

从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。

`script` 参数是一段 Lua 5.1 脚本程序，它会被运行在 Redis 服务器上下文中，这段脚本不必(也不应该)定义为一个 Lua 函数。

`numkeys` 参数用于指定键名参数的个数。

键名参数 `key [key ...]` 从 EVAL 的第三个参数开始算起，表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 `KEYS` 数组，用 `1` 为基址的形式访问(`KEYS[1]` ， `KEYS[2]` ，以此类推)。

在命令的最后，那些不是键名参数的附加参数 `arg [arg ...]` ，可以在 Lua 中通过全局变量 `ARGV` 数组访问，访问的形式和 `KEYS` 变量类似(`ARGV[1]` 、 `ARGV[2]` ，诸如此类)。

上面这几段长长的说明可以用一个简单的例子来概括：

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first
1) "key1"
2) "key2"
3) "first"
4) "second"
```

其中 `"return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"` 是被求值的 Lua 脚本，数字 `2` 指定了键名参数的数量， `key1` 和 `key2` 是键名参数，分别使用 `KEYS[1]` 和 `KEYS[2]` 访问，而最后的 `first` 和 `second` 则是附加参数，可以通过 `ARGV[1]` 和 `ARGV[2]` 访问它们。

在 Lua 脚本中，可以使用两个不同函数来执行 Redis 命令，它们分别是：

- `redis.call()`
- `redis.pcall()`

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误，在后面的『错误处理』部分会讲到这一点。

`redis.call()` 和 `redis.pcall()` 两个函数的参数可以是任何格式良好(well formed)的 Redis 命令：

```
> eval "return redis.call('set','foo','bar')" 0
OK
```

需要注意的是，上面这段脚本的确实现了将键 `foo` 的值设为 `bar` 的目的，但是，它违反了 EVAL 命令的语义，因为脚本里使用的所有键都应该由 `KEYS` 数组来传递，就像这样：

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
```

要求使用正确的形式来传递键(key)是有原因的，因为不仅仅是 EVAL 这个命令，所有的 Redis 命令，在执行之前都会被分析，籍此来确定命令会对哪些键进行操作。

因此，对于 EVAL 命令来说，必须使用正确的形式来传递键，才能确保分析工作正确地执行。除此之外，使用正确的形式来传递键还有很多其他好处，它的一个特别重要的用途就是确保 Redis 集群可以将你的请求发送到正确的集群节点。(对 Redis 集群的工作还在进行当中，但是脚本功能被设计成可以与集群功能保持兼容。)不过，这条规矩并不是强制性的，从而使得用户有机会滥用(abuse) Redis 单实例配置(single instance configuration)，代价是这样写出的脚本不能被 Redis 集群所兼容。

在 Lua 数据类型和 Redis 数据类型之间转换

当 Lua 通过 `call()` 或 `pcall()` 函数执行 Redis 命令的时候，命令的返回值会被转换成 Lua 数据结构。同样地，当 Lua 脚本在 Redis 内置的解释器里运行时，Lua 脚本的返回值也会被转换成 Redis 协议(protocol)，然后由 EVAL 将值返回给客户端。

数据类型之间的转换遵循这样一个设计原则：如果将一个 Redis 值转换成 Lua 值，之后再将转换所得的 Lua 值转换回 Redis 值，那么这个转换所得的 Redis 值应该和最初时的 Redis 值一样。

换句话说， Lua 类型和 Redis 类型之间存在着一一对应的转换关系。

以下列出的是详细的转换规则：

从 Redis 转换到 Lua ：

- Redis integer reply -> Lua number / Redis 整数转换成 Lua 数字
- Redis bulk reply -> Lua string / Redis bulk 回复转换成 Lua 字符串
- Redis multi bulk reply -> Lua table (may have other Redis data types nested) / Redis 多条 bulk 回复转换成 Lua 表，表内可能有其他别的 Redis 数据类型
- Redis status reply -> Lua table with a single ok field containing the status / Redis 状态回复转换成 Lua 表，表内的 `ok` 域包含了状态信息
- Redis error reply -> Lua table with a single err field containing the error / Redis 错误回复转换成 Lua 表，表内的 `err` 域包含了错误信息

- Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type / Redis 的 Nil 回复和 Nil 多条回复转换成 Lua 的布尔值 `false`

从 Lua 转换到 Redis：

- Lua number -> Redis integer reply / Lua 数字转换成 Redis 整数
- Lua string -> Redis bulk reply / Lua 字符串转换成 Redis bulk 回复
- Lua table (array) -> Redis multi bulk reply / Lua 表(数组)转换成 Redis 多条 bulk 回复
- Lua table with a single ok field -> Redis status reply / 一个带单个 `ok` 域的 Lua 表，转换成 Redis 状态回复
- Lua table with a single err field -> Redis error reply / 一个带单个 `err` 域的 Lua 表，转换成 Redis 错误回复
- Lua boolean false -> Redis Nil bulk reply / Lua 的布尔值 `false` 转换成 Redis 的 Nil bulk 回复

从 Lua 转换到 Redis 有一条额外的规则，这条规则没有和它对应的从 Redis 转换到 Lua 的规则：

- Lua boolean true -> Redis integer reply with value of 1 / Lua 布尔值 `true` 转换成 Redis 整数回复中的 `1`

以下是几个类型转换的例子：

```
> eval "return 10" 0
(integer) 10

> eval "return {1,2,{3,'Hello World!'}}" 0
1) (integer) 1
2) (integer) 2
3) 1) (integer) 3
   2) "Hello World!"

> eval "return redis.call('get','foo')" 0
"bar"
```

在上面的三个代码示例里，前两个演示了如何将 Lua 值转换成 Redis 值，最后一个例子更复杂一些，它演示了一个将 Redis 值转换成 Lua 值，然后再将 Lua 值转换成 Redis 值的类型转过程。

脚本的原子性

Redis 使用单个 Lua 解释器去运行所有脚本，并且， Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 *MULTI* / *EXEC* 包围的事务很类似。在其他别的客户端看来，脚本的效果(effect)要么是不可见的(not visible)，要么就是已完成的(already completed)。

另一方面，这也意味着，执行一个运行缓慢的脚本并不是一个好主意。写一个跑得很快很顺溜的脚本并不难，因为脚本的运行开销(overhead)非常少，但是当你不得不使用一些跑得比较慢的脚本时，请小心，因为当这些蜗牛脚本在慢吞吞地运行的时候，其他客户端会因为服务器正忙而无法执行命令。

错误处理

前面的命令介绍部分说过， `redis.call()` 和 `redis.pcall()` 的唯一区别在于它们对错误处理的不同。

当 `redis.call()` 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，错误的输出信息会说明错误造成的原因：

```
redis> lpush foo a
(integer) 1

redis> eval "return redis.call('get', 'foo')" 0
(error) ERR Error running script (call to f_282297a0228f48cd3fc6a55
```

和 `redis.call()` 不同， `redis.pcall()` 出错时并不引发(raise)错误，而是返回一个带 `err` 域的 Lua 表(table)，用于表示错误：

```
redis 127.0.0.1:6379> EVAL "return redis.pcall('get', 'foo')" 0
(error) ERR Operation against a key holding the wrong kind of value
```

带宽和 EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)。

EVALSHA 命令的表现如下：

- 如果服务器还记得给定的 SHA1 校验和所指定的脚本，那么执行这个脚本
- 如果服务器不记得给定的 SHA1 校验和所指定的脚本，那么它返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA

以下是示例：

```
> set foo bar
OK

> eval "return redis.call('get','foo')" 0
"bar"

> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0
"bar"

> evalsha ffffffffffffffffffffffffffffffffffffffff 0
(error) `NOSCRIPT` No matching script. Please use [EVAL](/commands/
```

客户端库的底层实现可以一直乐观地使用 EVALSHA 来代替 EVAL ，并期望着要使用的脚本已经保存在服务器上了，只有当 `NOSCRIPT` 错误发生时，才使用 EVAL 命令重新发送脚本，这样就可以最大限度地节省带宽。

这也说明了执行 EVAL 命令时，使用正确的格式来传递键名参数和附加参数的重要性：因为如果将参数硬写在脚本中，那么每次当参数改变的时候，都要重新发送脚本，即使脚本的主体并没有改变，相反，通过使用正确的格式来传递键名参数和附加参数，就可以在脚本主体不变的情况下，直接使用 EVALSHA 命令对脚本进行复用，免去了无谓的带宽消耗。

脚本缓存

Redis 保证所有被运行过的脚本都会被永久保存在脚本缓存当中，这意味着，当 EVAL 命令在一个 Redis 实例上成功执行某个脚本之后，随后针对这个脚本的所有 EVALSHA 命令都会成功执行。

刷新脚本缓存的唯一办法是显式地调用 `SCRIPT FLUSH` 命令，这个命令会清空运行过的所有脚本的缓存。通常只有在云计算环境中，Redis 实例被改作其他客户或者别的应用程序的实例时，才会执行这个命令。

缓存可以长时间储存而不产生内存问题的原因是，它们的体积非常小，而且数量也非常少，即使脚本在概念上类似于实现一个新命令，即使在一个大规模的程序里有成百上千的脚本，即使这些脚本会经常修改，即便如此，储存这些脚本的内存仍然是微不足道的。

事实上，用户会发现 Redis 不移除缓存中的脚本实际上是一个好主意。比如说，对于一个和 Redis 保持持久化链接(persistent connection)的程序来说，它可以确信，执行过一次的脚本会一直保留在内存当中，因此它可以在流水线中使用 EVALSHA 命令而不必担心因为找不到所需的脚本而产生错误(稍候我们会看到在流水线中执行脚本的相关问题)。

SCRIPT 命令

Redis 提供了以下几个 SCRIPT 命令，用于对脚本子系统(scripting subsystem)进行控制：

- *SCRIPT FLUSH* ：清除所有脚本缓存
- *SCRIPT EXISTS* ：根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
- *SCRIPT LOAD* ：将一个脚本装入脚本缓存，但并不立即运行它
- *SCRIPT KILL* ：杀死当前正在运行的脚本

纯函数脚本

在编写脚本方面，一个重要的要求就是，脚本应该被写成纯函数(pure function)。

也就是说，脚本应该具有以下属性：

- 对于同样的数据集输入，给定相同的参数，脚本执行的 Redis 写命令总是相同的。脚本执行的操作不能依赖于任何隐藏(非显式)数据，不能依赖于脚本在执行过程中、或脚本在不同执行时期之间可能变更的状态，并且它也不能依赖于任何来自 I/O 设备的外部输入。

使用系统时间(system time)，调用像 *RANDOMKEY* 那样的随机命令，或者使用 Lua 的随机数生成器，类似以上的这些操作，都会造成脚本的求值无法每次都得出同样的结果。

为了确保脚本符合上面所说的属性， Redis 做了以下工作：

- Lua 没有访问系统时间或者其他内部状态的命令
- Redis 会返回一个错误，阻止这样的脚本运行： 这些脚本在执行随机命令之后(比如 *RANDOMKEY* 、 *SRANDMEMBER* 或 *TIME* 等)，还会执行可以修改数据集的 Redis 命令。如果脚本只是执行只读操作，那么就没有这一限制。注意，随机命令并不一定就指那些带 RAND 字眼的命令，任何带有非确定性的命令都会被认为是随机命令，比如 *TIME* 命令就是这方面的一个很好的例子。
- 每当从 Lua 脚本中调用那些返回无序元素的命令时，执行命令所得的数据在返回给 Lua 之前会先执行一个静默(slient)的字典序排序(lexicographical sorting)。举个例子，因为 Redis 的 Set 保存的是无序的元素，所以在 Redis 命令行客户端中直接执行 *SMEMBERS* ，返回的元素是无序的，但是，假如在脚本中执行 `redis.call("smembers", KEYS[1])` ，那么返回的总是排过序的元素。
- 对 Lua 的伪随机数生成函数 `math.random` 和 `math.randomseed` 进行修改，使得每次在运行新脚本的时候，总是拥有同样的 seed 值。这意味着，每次运行脚本时，只要不使用 `math.randomseed` ，那么 `math.random` 产生的随机数序列总是相同的。

尽管有那么多的限制，但用户还是可以用一个简单的技巧写出带随机行为的脚本(如果他们需要的话)。

假设现在我们要编写一个 Redis 脚本，这个脚本从列表中弹出 N 个随机数。一个 Ruby 写的例子如下：

```
require 'rubygems'
require 'redis'

r = Redis.new

RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,[:mylist],[10,rand(2**32)])
```

这个程序每次运行都会生成带有以下元素的列表：

```
> lrange mylist 0 -1
1) "0.74509509873814"
2) "0.87390407681181"
3) "0.36876626981831"
4) "0.6921941534114"
5) "0.7857992587545"
6) "0.57730350670279"
7) "0.87046522734243"
8) "0.09637165539729"
9) "0.74990198051087"
10) "0.17082803611217"
```

上面的 Ruby 程序每次都只生成同样的列表，用途并不是太大。那么，该怎样修改这个脚本，使得它仍然是一个纯函数(符合 Redis 的要求)，但是每次调用都可以产生不同的随机元素呢？

一个简单的办法是，为脚本添加一个额外的参数，让这个参数作为 Lua 的随机数生成器的 seed 值，这样的话，只要给脚本传入不同的 seed ，脚本就会生成不同的列表元素。

以下是修改后的脚本：

```
RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    math.randomseed(tonumber(ARGV[2]))
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,1,:mylist,10,rand(2**32))
```

尽管对于同样的 seed ，上面的脚本产生的列表元素是一样的(因为它是一个纯函数)，但是只要每次在执行脚本的时候传入不同的 seed ，我们就可以得到带有不同随机元素的列表。

Seed 会在复制(replication link)和写 AOF 文件时作为一个参数来传播，保证在载入 AOF 文件或附属节点(slave)处理脚本时， seed 仍然可以及时得到更新。

注意，Redis 实现保证 `math.random` 和 `math.randomseed` 的输出和运行 Redis 的系统架构无关，无论是 32 位还是 64 位系统，无论是小端(little endian)还是大端(big endian)系统，这两个函数的输出总是相同的。

全局变量保护

为了防止不必要的数据泄漏进 Lua 环境， Redis 脚本不允许创建全局变量。如果一个脚本需要在多次执行之间维持某种状态，它应该使用 Redis key 来进行状态保存。

企图在脚本中访问一个全局变量(不论这个变量是否存在)将引起脚本停止， EVAL 命令会返回一个错误：

```
redis 127.0.0.1:6379> eval 'a=10' 0
(error) ERR Error running script (call to f_933044db579a2f8fd45d80
```

Lua 的 debug 工具，或者其他设施，比如打印（alter）用于实现全局保护的 meta table ，都可以用于实现全局变量保护。

实现全局变量保护并不难，不过有时候还是会不小心而为之。一旦用户在脚本中混入了 Lua 全局状态，那么 AOF 持久化和复制（replication）都会无法保证，所以，请不要使用全局变量。

避免引入全局变量的一个诀窍是：将脚本中用到的所有变量都使用 `local` 关键字定义为局部变量。

库

Redis 内置的 Lua 解释器加载了以下 Lua 库：

- `base`
- `table`
- `string`
- `math`
- `debug`
- `cjson`
- `cmsgpack`

其中 `cjson` 库可以让 Lua 以非常快的速度处理 JSON 数据，除此之外，其他别的都是 Lua 的标准库。

每个 Redis 实例都保证会加载上面列举的库，从而确保每个 Redis 脚本的运行环境都是相同的。

使用脚本散发 **Redis** 日志

在 Lua 脚本中，可以通过调用 `redis.log` 函数来写 Redis 日志(log)：

```
redis.log(loglevel, message)
```

其中， `message` 参数是一个字符串，而 `loglevel` 参数可以是以下任意一个值：

- `redis.LOG_DEBUG`
- `redis.LOG_VERBOSE`
- `redis.LOG_NOTICE`
- `redis.LOG_WARNING`

上面的这些等级(level)和标准 Redis 日志的等级相对应。

对于脚本散发(emit)的日志，只有那些和当前 Redis 实例所设置的日志等级相同或更高级的日志才会被散发。

以下是一个日志示例：

```
redis.log(redis.LOG_WARNING, "Something is wrong with this script."
```

执行上面的函数会产生这样的信息：

```
[32343] 22 Mar 15:21:39 # Something is wrong with this script.
```

沙箱(sandbox)和最大执行时间

脚本应该仅仅用于传递参数和对 Redis 数据进行处理，它不应该尝试去访问外部系统(比如文件系统)，或者执行任何系统调用。

除此之外，脚本还有一个最大执行时间限制，它的默认值是 5 秒钟，一般正常运作的脚本通常可以在几分之几毫秒之内完成，花不了那么多时间，这个限制主要是为了防止因编程错误而造成的无限循环而设置的。

最大执行时间的长短由 `lua-time-limit` 选项来控制(以毫秒为单位)，可以通过编辑 `redis.conf` 文件或者使用 *CONFIG GET* 和 *CONFIG SET* 命令来修改它。

当一个脚本达到最大执行时间的时候，它并不会自动被 Redis 结束，因为 Redis 必须保证脚本执行的原子性，而中途停止脚本的运行意味着可能会留下未处理完的数据在数据集(data set)里面。

因此，当脚本运行的时间超过最大执行时间后，以下动作会被执行：

- Redis 记录一个脚本正在超时运行
- Redis 开始重新接受其他客户端的命令请求，但是只有 `SCRIPT KILL` 和 `SHUTDOWN NOSAVE` 两个命令会被处理，对于其他命令请求， Redis 服务器只是简单地返回 `BUSY` 错误。
- 可以使用 `SCRIPT KILL` 命令将一个仅执行只读命令的脚本杀死，因为只读命令并不修改数据，因此杀死这个脚本并不破坏数据的完整性
- 如果脚本已经执行过写命令，那么唯一允许执行的操作就是 `SHUTDOWN NOSAVE` ，它通过停止服务器来阻止当前数据集写入磁盘

流水线(pipeline)上下文(context)中的 EVALSHA

在流水线请求的上下文中使用 EVALSHA 命令时，要特别小心，因为在流水线中，必须保证命令的执行顺序。

一旦在流水线中因为 EVALSHA 命令而发生 NOSCRIPT 错误，那么这个流水线就再也没有办法重新执行了，否则的话，命令的执行顺序就会被打乱。

为了防止出现以上所说的问题，客户端库实现应该实施以下的其中一项措施：

- 总是在流水线中使用 EVAL 命令
- 检查流水线中要用到的所有命令，找到其中的 EVAL 命令，并使用 *SCRIPT EXISTS* 命令检查要用到的脚本是不是全都已经保存在缓存里面了。如果所需的全部脚本都可以在缓存里找到，那么就可以放心地将所有 EVAL 命令改成 EVALSHA 命令，否则的话，就要在流水线的顶端(top)将缺少的脚本用 *SCRIPT LOAD* 命令加上去。

可用版本：

>= 2.6.0

时间复杂度：

EVAL 和 EVALSHA 可以在 O(1) 复杂度内找到要被执行的脚本，其余的复杂度取决于执行的脚本本身。

EVALSHA

EVALSHA sha1 numkeys key [key ...] arg [arg ...]

根据给定的 sha1 校验码，对缓存在服务器中的脚本进行求值。

将脚本缓存到服务器的操作可以通过 *SCRIPT LOAD* 命令进行。

这个命令的其他地方，比如参数的传入方式，都和 *EVAL* 命令一样。

可用版本：

>= 2.6.0

时间复杂度：

根据脚本的复杂度而定。

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA "232fd51614574cf0867b83d384a5e898cfd24e5a" 0
"hello moto"
```

SCRIPT EXISTS

SCRIPT EXISTS script [script ...]

给定一个或多个脚本的 SHA1 校验和，返回一个包含 `0` 和 `1` 的列表，表示校验和所指定的脚本是否已经被保存在缓存当中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为给定的 SHA1 校验和的数量。

返回值：

一个列表，包含 `0` 和 `1` ，前者表示脚本不存在于缓存，后者表示脚本已经在缓存里面了。列表中的元素和给定的 SHA1 校验和保持对应关系，比如列表的第三个元素的值就表示第三个 SHA1 校验和所指定的脚本在缓存中的状态。

```
redis> SCRIPT LOAD "return 'hello moto'"    # 载入一个脚本
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 1

redis> SCRIPT FLUSH     # 清空缓存
OK

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 0
```

SCRIPT FLUSH

SCRIPT FLUSH

清除所有 Lua 脚本缓存。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

复杂度：

O(N)， `N` 为缓存中脚本的数量。

返回值：

总是返回 `OK`

```
redis> SCRIPT FLUSH
OK
```

SCRIPT KILL

SCRIPT KILL

杀死当前正在运行的 Lua 脚本，当且仅当这个脚本没有执行过任何写操作时，这个命令才生效。

这个命令主要用于终止运行时间过长的脚本，比如一个因为 BUG 而发生无限 loop 的脚本，诸如此类。

SCRIPT KILL 执行之后，当前正在运行的脚本会被杀死，执行这个脚本的客户端会从 *EVAL* 命令的阻塞当中退出，并收到一个错误作为返回值。

另一方面，假如当前正在运行的脚本已经执行过写操作，那么即使执行 SCRIPT KILL ，也无法将它杀死，因为这是违反 Lua 脚本的原子性执行原则的。在这种情况下，唯一可行的办法是使用 `SHUTDOWN NOSAVE` 命令，通过停止整个 Redis 进程来停止脚本的运行，并防止不完整(half-written)的信息被写入数据库中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行成功返回 `OK` ，否则返回一个错误。

```
# 没有脚本在执行时

redis> SCRIPT KILL
(error) ERR No scripts in execution right now.

# 成功杀死脚本时

redis> SCRIPT KILL
OK
(1.30s)

# 尝试杀死一个已经执行过写操作的脚本，失败

redis> SCRIPT KILL
(error) ERR Sorry the script already executed write commands agains
(1.69s)
```

以下是脚本被杀死之后，返回给执行脚本的客户端的错误：

```
redis> EVAL "while true do end" 0
(error) ERR Error running script (call to f_694a5fe1ddb97a4c6a1bf29
(5.00s)
```

SCRIPT LOAD

SCRIPT LOAD script

将脚本 `script` 添加到脚本缓存中，但并不立即执行这个脚本。

EVAL 命令也会将脚本添加到脚本缓存中，但是它会立即对输入的脚本进行求值。

如果给定的脚本已经在缓存里面了，那么不做动作。

在脚本被加入到缓存之后，通过 EVALSHA 命令，可以使用脚本的 SHA1 校验和来调用这个脚本。

脚本可以在缓存中保留无限长的时间，直到执行 *SCRIPT FLUSH* 为止。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为脚本的长度(以字节为单位)。

返回值：

给定 `script` 的 SHA1 校验和

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA 232fd51614574cf0867b83d384a5e898cfd24e5a 0
"hello moto"
```

Connection（连接）

AUTH

AUTH password

通过设置配置文件中 `requirepass` 项的值(使用命令 `CONFIG SET requirepass password`)，可以使用密码来保护 Redis 服务器。

如果开启了密码保护的话，在每次连接 Redis 服务器之后，就要使用 `AUTH` 命令解锁，解锁之后才能使用其他 Redis 命令。

如果 `AUTH` 命令给定的密码 `password` 和配置文件中的密码相符的话，服务器会返回 `OK` 并开始接受命令输入。

另一方面，假如密码不匹配的话，服务器将返回一个错误，并要求客户端需重新输入密码。

Warning

因为 Redis 高性能的特点，在很短时间内尝试猜测非常多个密码是有可能的，因此请确保使用的密码足够复杂和足够长，以免遭受密码猜测攻击。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

密码匹配时返回 `OK` ，否则返回一个错误。

```
# 设置密码

redis> CONFIG SET requirepass secret_password   # 将密码设置为 secret
OK

redis> QUIT                                     # 退出再连接，让新密码

[huangz@mypad]$ redis

redis> PING                                     # 未验证密码，操作被拒
(error) ERR operation not permitted

redis> AUTH wrong_password_testing              # 尝试输入错误的密码
(error) ERR invalid password

redis> AUTH secret_password                     # 输入正确的密码
OK

redis> PING                                     # 密码验证成功，可以正
PONG

# 清空密码

redis> CONFIG SET requirepass ""   # 通过将密码设为空字符来清空密码
OK

redis> QUIT

$ redis                            # 重新进入客户端

redis> PING                        # 执行命令不再需要密码，清空密码操作成
PONG
```

ECHO

ECHO message

打印一个特定的信息 `message` ，测试时使用。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`message` 自身。

```
redis> ECHO "Hello Moto"
"Hello Moto"

redis> ECHO "Goodbye Moto"
"Goodbye Moto"
```

PING

PING

使用客户端向 Redis 服务器发送一个 `PING` ，如果服务器运作正常的话，会返回一个 `PONG` 。

通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

如果连接正常就返回一个 `PONG` ，否则返回一个连接错误。

```
# 客户端和服务器连接正常

redis> PING
PONG

# 客户端和服务器连接不正常(网络不正常或服务器未能正常运行)

redis 127.0.0.1:6379> PING
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

QUIT

QUIT

请求服务器关闭与当前客户端的连接。

一旦所有等待中的回复(如果有的话)顺利写入到客户端，连接就会被关闭。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` (但是不会被打印显示，因为当时 Redis-cli 已经退出)。

```
$ redis

redis> QUIT

$
```

SELECT

SELECT index

切换到指定的数据库，数据库索引号 `index` 用数字值指定，以 `0` 作为起始索引值。

默认使用 `0` 号数据库。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`OK`

```
redis> SET db_number 0          # 默认使用 0 号数据库
OK

redis> SELECT 1                 # 使用 1 号数据库
OK

redis[1]> GET db_number         # 已经切换到 1 号数据库，注意 Redis 现在
(nil)

redis[1]> SET db_number 1
OK

redis[1]> GET db_number
"1"

redis[1]> SELECT 3              # 再切换到 3 号数据库
OK

redis[3]>                       # 提示符从 [1] 改变成了 [3]
```

Server（服务器）

BGREWRITEAOF

BGREWRITEAOF

执行一个 AOF文件 重写操作。重写会创建一个当前 AOF 文件的体积优化版本。

即使 BGREWRITEAOF 执行失败，也不会有任何数据丢失，因为旧的 AOF 文件在 BGREWRITEAOF 成功之前不会被修改。

重写操作只会在没有其他持久化工作在后台执行时被触发，也就是说：

- 如果 Redis 的子进程正在执行快照的保存工作，那么 AOF 重写的操作会被预定(scheduled)，等到保存工作完成之后再执行 AOF 重写。在这种情况下，BGREWRITEAOF 的返回值仍然是 `OK` ，但还会加上一条额外的信息，说明 BGREWRITEAOF 要等到保存操作完成之后才能执行。在 Redis 2.6 或以上的版本，可以使用 *INFO* 命令查看 BGREWRITEAOF 是否被预定。
- 如果已经有别的 AOF 文件重写在执行，那么 BGREWRITEAOF 返回一个错误，并且这个新的 BGREWRITEAOF 请求也不会被预定到下次执行。

从 Redis 2.4 开始， AOF 重写由 Redis 自行触发， BGREWRITEAOF 仅仅用于手动触发重写操作。

请移步 持久化文档(英文) 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要追加到 AOF 文件中的数据数量。

返回值：

反馈信息。

```
redis> BGREWRITEAOF
Background append only file rewriting started
```

BGSAVE

在后台异步(Asynchronously)保存当前数据库的数据到磁盘。

BGSAVE 命令执行之后立即返回 `OK` ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。

客户端可以通过 *LASTSAVE* 命令查看相关信息，判断 BGSAVE 命令是否执行成功。

请移步 持久化文档 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

反馈信息。

```
redis> BGSAVE
Background saving started
```

CLIENT GETNAME

CLIENT GETNAME

返回 *CLIENT SETNAME* 命令为连接设置的名字。

因为新创建的连接默认是没有名字的， 对于没有名字的连接， *CLIENT GETNAME* 返回空白回复。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

如果连接没有设置名字，那么返回空白回复；如果有设置名字，那么返回名字。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"
```

CLIENT KILL

CLIENT KILL ip:port

关闭地址为 `ip:port` 的客户端。

`ip:port` 应该和 *CLIENT LIST* 命令输出的其中一行匹配。

因为 Redis 使用单线程设计，所以当 Redis 正在执行命令的时候，不会有客户端被断开连接。

如果要被断开连接的客户端正在执行命令，那么当这个命令执行之后，在发送下一个命令的时候，它就会收到一个网络错误，告知它自身的连接已被关闭。

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为已连接的客户端数量。

返回值

当指定的客户端存在，且被成功关闭时，返回 `OK` 。

```
# 列出所有已连接客户端

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43501 fd=5 age=10 idle=0 flags=N db=0 sub=0 psub=0 r

# 杀死当前客户端的连接

redis 127.0.0.1:6379> CLIENT KILL 127.0.0.1:43501
OK

# 之前的连接已经被关闭，CLI 客户端又重新建立了连接
# 之前的端口是 43501 ，现在是 43504

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43504 fd=5 age=0 idle=0 flags=N db=0 sub=0 psub=0 mu
```

CLIENT LIST

CLIENT LIST

以人类可读的格式，返回所有连接到服务器的客户端信息和统计数据。

```
redis> CLIENT LIST
addr=127.0.0.1:43143 fd=6 age=183 idle=0 flags=N db=0 sub=0 psub=0
addr=127.0.0.1:43163 fd=5 age=35 idle=15 flags=N db=0 sub=0 psub=0
addr=127.0.0.1:43167 fd=7 age=24 idle=6 flags=N db=0 sub=0 psub=0 r
```

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为连接到服务器的客户端数量。

返回值

命令返回多行字符串，这些字符串按以下形式被格式化：

- 每个已连接客户端对应一行（以 `LF` 分割）
- 每行字符串由一系列 `属性=值` 形式的域组成，每个域之间以空格分开

以下是域的含义：

- `addr` ：客户端的地址和端口
- `fd` ：套接字所使用的文件描述符
- `age` ：以秒计算的已连接时长
- `idle` ：以秒计算的空闲时长
- `flags` ：客户端 flag （见下文）
- `db` ：该客户端正在使用的数据库 ID
- `sub` ：已订阅频道的数量
- `psub` ：已订阅模式的数量
- `multi` ：在事务中被执行的命令数量
- `qbuf` ：查询缓冲区的长度（字节为单位， `0` 表示没有分配查询缓冲区）
- `qbuf-free` ：查询缓冲区剩余空间的长度（字节为单位， `0` 表示没有剩余空间）
- `obl` ：输出缓冲区的长度（字节为单位， `0` 表示没有分配输出缓冲区）
- `oll` ：输出列表包含的对象数量（当输出缓冲区没有剩余空间时，命令回复会以字符串对象的形式被入队到这个队列里）
- `omem` ：输出缓冲区和输出列表占用的内存总量
- `events` ：文件描述符事件（见下文）
- `cmd` ：最近一次执行的命令

客户端 flag 可以由以下部分组成：

- `O` ：客户端是 MONITOR 模式下的附属节点（slave）
- `S` ：客户端是一般模式下（normal）的附属节点
- `M` ：客户端是主节点（master）
- `x` ：客户端正在执行事务
- `b` ：客户端正在等待阻塞事件
- `i` ：客户端正在等待 VM I/O 操作（已废弃）
- `d` ：一个受监视（watched）的键已被修改， `EXEC` 命令将失败
- `c` : 在将回复完整地写出之后，关闭链接
- `u` : 客户端未被阻塞（unblocked）
- `A` : 尽可能快地关闭连接
- `N` : 未设置任何 flag

文件描述符事件可以是：

- `r` : 客户端套接字（在事件 loop 中）是可读的（readable）
- `w` : 客户端套接字（在事件 loop 中）是可写的（writeable）

Note

为了 debug 的需要，经常会对域进行添加和删除，一个安全的 Redis 客户端应该可以对 `CLIENT LIST` 的输出进行相应的处理（parse），比如忽略不存在的域，跳过未知域，诸如此类。

CLIENT SETNAME

CLIENT SETNAME connection-name

为当前连接分配一个名字。

这个名字会显示在 *CLIENT LIST* 命令的结果中， 用于识别当前正在与服务器进行连接的客户端。

举个例子， 在使用 Redis 构建队列（queue）时， 可以根据连接负责的任务（role）， 为信息生产者（producer）和信息消费者（consumer）分别设置不同的名字。

名字使用 Redis 的字符串类型来保存， 最大可以占用 512 MB 。 另外， 为了避免和 *CLIENT LIST* 命令的输出格式发生冲突， 名字里不允许使用空格。

要移除一个连接的名字， 可以将连接的名字设为空字符串 `""` 。

使用 *CLIENT GETNAME* 命令可以取出连接的名字。

新创建的连接默认是没有名字的。

Tip

在 Redis 应用程序发生连接泄漏时，为连接设置名字是一种很好的 debug 手段。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

设置成功时返回 `OK` 。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"

# 在客户端列表中查看

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:36851
fd=5
name=hello-world-connection     # <- 名字
age=51
...

# 清除名字

redis 127.0.0.1:6379> CLIENT SETNAME        # 只用空格是不行的！
(error) ERR Syntax error, try CLIENT (LIST | KILL ip:port)

redis 127.0.0.1:6379> CLIENT SETNAME ""     # 必须双引号显示包围
OK

redis 127.0.0.1:6379> CLIENT GETNAME        # 清除完毕
(nil)
```

CONFIG GET

CONFIG GET parameter

CONFIG GET 命令用于取得运行中的 Redis 服务器的配置参数(configuration parameters)，在 Redis 2.4 版本中， 有部分参数没有办法用 `CONFIG GET` 访问，但是在最新的 Redis 2.6 版本中，所有配置参数都已经可以用 `CONFIG GET` 访问了。

CONFIG GET 接受单个参数 `parameter` 作为搜索关键字，查找所有匹配的配置参数，其中参数和值以“键-值对”(key-value pairs)的方式排列。

比如执行 `CONFIG GET s*` 命令，服务器就会返回所有以 `s` 开头的配置参数及参数的值：

```
redis> CONFIG GET s*
1) "save"                       # 参数名：save
2) "900 1 300 10 60 10000"      # save 参数的值
3) "slave-serve-stale-data"     # 参数名： slave-serve-stale-data
4) "yes"                        # slave-serve-stale-data 参数的值
5) "set-max-intset-entries"     # ...
6) "512"
7) "slowlog-log-slower-than"
8) "1000"
9) "slowlog-max-len"
10) "1000"
```

如果你只是寻找特定的某个参数的话，你当然也可以直接指定参数的名字：

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

使用命令 `CONFIG GET *` ，可以列出 `CONFIG GET` 命令支持的所有参数：

```
redis> CONFIG GET *
1) "dir"
2) "/var/lib/redis"
3) "dbfilename"
4) "dump.rdb"
5) "requirepass"
6) (nil)
7) "masterauth"
8) (nil)
9) "maxmemory"
10) "0"
11) "maxmemory-policy"
12) "volatile-lru"
13) "maxmemory-samples"
14) "3"
15) "timeout"
16) "0"
17) "appendonly"
18) "no"
# ...
49) "loglevel"
50) "verbose"
```

所有被 `CONFIG SET` 所支持的配置参数都可以在配置文件 redis.conf 中找到，不过 `CONFIG GET` 和 `CONFIG SET` 使用的格式和 redis.conf 文件所使用的格式有以下两点不同：

- `10kb` 、 `2gb` 这些在配置文件中所使用的储存单位缩写，不可以用在 `CONFIG` 命令中， `CONFIG SET` 的值只能通过数字值显式地设定。像 `CONFIG SET xxx 1k` 这样的命令是错误的，正确的格式是 `CONFIG SET xxx 1000` 。
- `save` 选项在 redis.conf 中是用多行文字储存的，但在 `CONFIG GET` 命令中，它只打印一行文字。以下是 `save` 选项在 redis.conf 文件中的表示： save 900 1``save 300 10``save 60 10000 但是 `CONFIG GET` 命令的输出只有一行： redis> CONFIG GET save``1) "save"``2) "900 1 300 10 60 10 上面 `save` 参数的三个值表示：在 900 秒内最少有 1 个 key 被改动，或者 300 秒内最少有 10 个 key 被改动，又或者 60 秒内最少有 1000 个 key 被改动，以上三个条件随便满足一个，就触发一次保存操作。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

给定配置参数的值。

CONFIG RESETSTAT

CONFIG RESETSTAT

重置 *INFO* 命令中的某些统计数据，包括：

- Keyspace hits (键空间命中次数)
- Keyspace misses (键空间不命中次数)
- Number of commands processed (执行命令的次数)
- Number of connections received (连接服务器的次数)
- Number of expired keys (过期key的数量)
- Number of rejected connections (被拒绝的连接数量)
- Latest fork(2) time(最后执行 fork(2) 的时间)
- The `aof_delayed_fsync` counter(`aof_delayed_fsync` 计数器的值)

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
# 重置前

redis 127.0.0.1:6379> INFO
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:6
uptime_in_days:0
lru_clock:1205426

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0
```

```
# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
used_memory_peak:293424
used_memory_peak_human:286.55K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.01
used_cpu_user:0.00
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0

# 重置

redis 127.0.0.1:6379> CONFIG RESETSTAT
OK

# 重置后

redis 127.0.0.1:6379> INFO
```

```
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:134
uptime_in_days:0
lru_clock:1205438

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
used_memory_peak:330280
used_memory_peak_human:322.54K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:0
total_commands_processed:1
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
```

```
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.05
used_cpu_user:0.02
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0
```

CONFIG REWRITE

CONFIG REWRITE

CONFIG REWRITE 命令对启动 Redis 服务器时所指定的 `redis.conf` 文件进行改写： 因为 *CONFIG SET* 命令可以对服务器的当前配置进行修改， 而修改后的配置可能和 `redis.conf` 文件中所描述的配置不一样， *CONFIG REWRITE* 的作用就是通过尽可能少的修改， 将服务器当前所使用的配置记录到 `redis.conf` 文件中。

重写会以非常保守的方式进行：

- 原有 `redis.conf` 文件的整体结构和注释会被尽可能地保留。
- 如果一个选项已经存在于原有 `redis.conf` 文件中， 那么对该选项的重写会在选项原本所在的位置（行号）上进行。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为默认值， 那么重写程序不会将这个选项添加到重写后的 `redis.conf` 文件中。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为非默认值， 那么这个选项将被添加到重写后的 `redis.conf` 文件的末尾。
- 未使用的行会被留白。 比如说， 如果你在原有 `redis.conf` 文件上设置了数个关于 `save` 选项的参数， 但现在你将这些 `save` 参数的一个或全部都关闭了， 那么这些不再使用的参数原本所在的行就会变成空白的。

即使启动服务器时所指定的 `redis.conf` 文件已经不再存在， *CONFIG REWRITE* 命令也可以重新构建并生成出一个新的 `redis.conf` 文件。

另一方面， 如果启动服务器时没有载入 `redis.conf` 文件， 那么执行 *CONFIG REWRITE* 命令将引发一个错误。

原子性重写

对 `redis.conf` 文件的重写是原子性的， 并且是一致的： 如果重写出错或重写期间服务器崩溃， 那么重写失败， 原有 `redis.conf` 文件不会被修改。 如果重写成功， 那么 `redis.conf` 文件为重写后的新文件。

可用版本

>= 2.8.0

返回值

一个状态值：如果配置重写成功则返回 `OK` ，失败则返回一个错误。

测试

以下是执行 *CONFIG REWRITE* 前，被载入到 Redis 服务器的 `redis.conf` 文件中关于 `appendonly` 选项的设置：

```
# ... 其他选项

appendonly no

# ... 其他选项
```

在执行以下命令之后：

```
127.0.0.1:6379> CONFIG GET appendonly           # appendonly 处于关
1) "appendonly"
2) "no"

127.0.0.1:6379> CONFIG SET appendonly yes       # 打开 appendonly
OK

127.0.0.1:6379> CONFIG GET appendonly
1) "appendonly"
2) "yes"

127.0.0.1:6379> CONFIG REWRITE                  # 将 appendonly 的修
OK
```

重写后的 `redis.conf` 文件中的 `appendonly` 选项将被改写：

```
# ... 其他选项

appendonly yes

# ... 其他选项
```

CONFIG SET

CONFIG SET parameter value

CONFIG SET 命令可以动态地调整 Redis 服务器的配置(configuration)而无须重启。

你可以使用它修改配置参数，或者改变 Redis 的持久化(Persistence)方式。

CONFIG SET 可以修改的配置参数可以使用命令 `CONFIG GET *` 来列出，所有被 CONFIG SET 修改的配置参数都会立即生效。

关于 CONFIG SET 命令的更多消息，请参见命令 *CONFIG GET* 的说明。

关于如何使用 CONFIG SET 命令修改 Redis 持久化方式，请参见 Redis Persistence 。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

当设置成功时返回 `OK` ，否则返回一个错误。

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1024"

redis> CONFIG SET slowlog-max-len 10086
OK

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "10086"
```

DBSIZE

DBSIZE

返回当前数据库的 key 的数量。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当前数据库的 key 的数量。

```
redis> DBSIZE
(integer) 5

redis> SET new_key "hello_moto"     # 增加一个 key 试试
OK

redis> DBSIZE
(integer) 6
```

DEBUG OBJECT

DEBUG OBJECT key

DEBUG OBJECT 是一个调试命令，它不应被客户端所使用。

查看 *OBJECT* 命令获取更多信息。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 存在时，返回有关信息。当 `key` 不存在时，返回一个错误。

```
redis> DEBUG OBJECT my_pc
Value at:0xb6838d20 refcount:1 encoding:raw serializedlength:9 lru

redis> DEBUG OBJECT your_mac
(error) ERR no such key
```

DEBUG SEGFAULT

DEBUG SEGFAULT

执行一个不合法的内存访问从而让 Redis 崩溃，仅在开发时用于 BUG 模拟。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

无

```
redis> DEBUG SEGFAULT
Could not connect to Redis at: Connection refused

not connected>
```

FLUSHALL

FLUSHALL

清空整个 Redis 服务器的数据(删除所有数据库的所有 key)。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

尚未明确

返回值：

总是返回 `OK` 。

```
redis> DBSIZE            # 0 号数据库的 key 数量
(integer) 9

redis> SELECT 1          # 切换到 1 号数据库
OK

redis[1]> DBSIZE         # 1 号数据库的 key 数量
(integer) 6

redis[1]> flushall       # 清空所有数据库的所有 key
OK

redis[1]> DBSIZE         # 不但 1 号数据库被清空了
(integer) 0

redis[1]> SELECT 0       # 0 号数据库(以及其他所有数据库)也一样
OK

redis> DBSIZE
(integer) 0
```

FLUSHDB

FLUSHDB

清空当前数据库中的所有 key。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
redis> DBSIZE    # 清空前的 key 数量
(integer) 4

redis> FLUSHDB
OK

redis> DBSIZE    # 清空后的 key 数量
(integer) 0
```

INFO

INFO [section]

以一种易于解释（parse）且易于阅读的格式，返回关于 Redis 服务器的各种信息和统计数值。

通过给定可选的参数 `section` ，可以让命令只返回某一部分的信息：

- `server` : 一般 Redis 服务器信息，包含以下域：

  > `redis_version` : Redis 服务器版本 > `redis_git_sha1` : Git SHA1 > `redis_git_dirty` : Git dirty flag > `os` : Redis 服务器的宿主操作系统 > `arch_bits` : 架构（32 或 64 位） > `multiplexing_api` : Redis 所使用的事件处理机制 > `gcc_version` : 编译 Redis 时所使用的 GCC 版本 > `process_id` : 服务器进程的 PID > `run_id` : Redis 服务器的随机标识符（用于 Sentinel 和集群） > `tcp_port` : TCP/IP 监听端口 > `uptime_in_seconds` : 自 Redis 服务器启动以来，经过的秒数 > `uptime_in_days` : 自 Redis 服务器启动以来，经过的天数 > * `lru_clock` : 以分钟为单位进行自增的时钟，用于 LRU 管理

- `clients` : 已连接客户端信息，包含以下域：

  > `connected_clients` : 已连接客户端的数量（不包括通过从属服务器连接的客户端） > `client_longest_output_list` : 当前连接的客户端当中，最长的输出列表 > `client_longest_input_buf` : 当前连接的客户端当中，最大输入缓存 > `blocked_clients` : 正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量

- `memory` : 内存信息，包含以下域：

  > `used_memory` : 由 Redis 分配器分配的内存总量，以字节（byte）为单位 > `used_memory_human` : 以人类可读的格式返回 Redis 分配的内存总量 > `used_memory_rss` : 从操作系统的角度，返回 Redis 已分配的内存总量（俗称常驻集大小）。这个值和 `top` 、 `ps` 等命令的输出一致。 > `used_memory_peak` : Redis 的内存消耗峰值（以字节为单位） > `used_memory_peak_human` : 以人类可读的格式返回 Redis 的内存消耗峰值 > `used_memory_lua` : Lua 引擎所使用的内存大小（以字节为单位） > `mem_fragmentation_ratio` : `used_memory_rss` 和 `used_memory` 之间的比率 > `mem_allocator` : 在编译时指定的， Redis 所使用的内存分配器。可以是 libc 、 jemalloc 或者 tcmalloc 。 > > 在理想情况下， `used_memory_rss` 的值应该只比 `used_memory` 稍微高一点儿。当 `rss > used` ，且两者的值相差较大时，表示存在（内部或外部的）内存碎片。内存碎片的比率可以通过 `mem_fragmentation_ratio` 的值看出。当 `used > rss` 时，表示 Redis 的部分内存被操作系统换出到交换空间了，在这种情况下，操作可能会产生明显的延迟。 > > Because Redis does not have control over how its allocations are mapped to memory pages, high

`used_memory_rss` is often the result of a spike in memory usage. > > 当 Redis 释放内存时，分配器可能会，也可能不会，将内存返还给操作系统。如果 Redis 释放了内存，却没有将内存返还给操作系统，那么 `used_memory` 的值可能和操作系统显示的 Redis 内存占用并不一致。查看 `used_memory_peak` 的值可以验证这种情况是否发生。

- `persistence` : `RDB` 和 `AOF` 的相关信息
- `stats` : 一般统计信息
- `replication` : 主/从复制信息
- `cpu` : CPU 计算量统计信息
- `commandstats` : Redis 命令统计信息
- `cluster` : Redis 集群信息
- `keyspace` : 数据库相关的统计信息

除上面给出的这些值以外，参数还可以是下面这两个：

- `all` : 返回所有信息
- `default` : 返回默认选择的信息

当不带参数直接调用 INFO 命令时，使用 `default` 作为默认参数。

Note

不同版本的 Redis 可能对返回的一些域进行了增加或删减。

因此，一个健壮的客户端程序在对 *INFO* 命令的输出进行分析时，应该能够跳过不认识的域，并且妥善地处理丢失不见的域。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

具体请参见下面的测试代码。

```
redis> INFO
# Server
redis_version:2.5.9
redis_git_sha1:473f3090
redis_git_dirty:0
os:Linux 3.3.7-1-ARCH i686
arch_bits:32
multiplexing_api:epoll
```

```
gcc_version:4.7.0
process_id:8104
run_id:bc9e20c6f0aac67d0d396ab950940ae4d1479ad1
tcp_port:6379
uptime_in_seconds:7
uptime_in_days:0
lru_clock:1680564

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:439304
used_memory_human:429.01K
used_memory_rss:13897728
used_memory_peak:401776
used_memory_peak_human:392.36K
used_memory_lua:20480
mem_fragmentation_ratio:31.64
mem_allocator:jemalloc-3.0.0

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1338011402
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
```

```
connected_slaves:0

# CPU
used_cpu_sys:0.03
used_cpu_user:0.01
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
```

LASTSAVE

LASTSAVE

返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

一个 UNIX 时间戳。

```
redis> LASTSAVE
(integer) 1324043588
```

MONITOR

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

总是返回 `OK` 。

```
127.0.0.1:6379> MONITOR
OK
# 以第一个打印值为例
# 1378822099.421623 是时间戳
# [0 127.0.0.1:56604] 中的 0 是数据库号码， 127... 是 IP 地址和端口
# "PING" 是被执行的命令
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Bana
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"
```

PSYNC

PSYNC <MASTER_RUN_ID> <OFFSET>

用于复制功能(replication)的内部命令。

更多信息请参考 复制（*Replication*） 文档。

可用版本：

>= 2.8.0

时间复杂度：

不明确

返回值：

不明确

```
127.0.0.1:6379> PSYNC ? -1
"REDIS0006\xfe\x00\x00\x02kk\x02vv\x00\x03msg\x05hello\xff\xc3\x96P
```

SAVE

SAVE

SAVE 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘。

一般来说，在生产环境很少执行 SAVE 操作，因为它会阻塞所有客户端，保存数据库的任务通常由 *BGSAVE* 命令异步地执行。然而，如果负责保存数据的后台子进程不幸出现问题时， SAVE 可以作为保存数据的最后手段来使用。

请参考文档： Redis 的持久化运作方式(英文) 以获取更多消息。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

保存成功时返回 `OK` 。

```
redis> SAVE
OK
```

SHUTDOWN

SHUTDOWN

SHUTDOWN 命令执行以下操作：

- 停止所有客户端
- 如果有至少一个保存点在等待，执行 *SAVE* 命令
- 如果 AOF 选项被打开，更新 AOF 文件
- 关闭 redis 服务器(server)

如果持久化被打开的话， SHUTDOWN 命令会保证服务器正常关闭而不丢失任何数据。

另一方面，假如只是单纯地执行 *SAVE* 命令，然后再执行 *QUIT* 命令，则没有这一保证 —— 因为在执行 *SAVE* 之后、执行 *QUIT* 之前的这段时间中间，其他客户端可能正在和服务器进行通讯，这时如果执行 *QUIT* 就会造成数据丢失。

SAVE 和 NOSAVE 修饰符

通过使用可选的修饰符，可以修改 SHUTDOWN 命令的表现。比如说：

- 执行 `SHUTDOWN SAVE` 会强制让数据库执行保存操作，即使没有设定(configure)保存点
- 执行 `SHUTDOWN NOSAVE` 会阻止数据库执行保存操作，即使已经设定有一个或多个保存点(你可以将这一用法看作是强制停止服务器的一个假想的 ABORT 命令)

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

执行失败时返回错误。执行成功时不返回任何信息，服务器和客户端的连接断开，客户端自动退出。

```
redis> PING
PONG

redis> SHUTDOWN

$

$ redis
Could not connect to Redis at: Connection refused
not connected>
```

SLAVEOF

`SLAVEOF host port`

SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为。

通过执行 `SLAVEOF host port` 命令，可以将当前服务器转变为指定服务器的从属服务器(slave server)。

如果当前服务器已经是某个主服务器(master server)的从属服务器，那么执行 `SLAVEOF host port` 将使当前服务器停止对旧主服务器的同步，丢弃旧数据集，转而开始对新主服务器进行同步。

另外，对一个从属服务器执行命令 `SLAVEOF NO ONE` 将使得这个从属服务器关闭复制功能，并从从属服务器转变回主服务器，原来同步所得的数据集不会被丢弃。

利用『 `SLAVEOF NO ONE` 不会丢弃同步所得数据集』这个特性，可以在主服务器失败的时候，将从属服务器用作新的主服务器，从而实现无间断运行。

可用版本：

>= 1.0.0

时间复杂度：

`SLAVEOF host port` ，O(N)， `N` 为要同步的数据数量。`SLAVEOF NO ONE` ， O(1) 。

返回值：

总是返回 `OK` 。

```
redis> SLAVEOF 127.0.0.1 6379
OK

redis> SLAVEOF NO ONE
OK
```

SLOWLOG

SLOWLOG subcommand [argument]

什么是 **SLOWLOG**

Slow log 是 Redis 用来记录查询执行时间的日志系统。

查询执行时间指的是不包括像客户端响应(talking)、发送回复等 IO 操作，而单单是执行一个查询命令所耗费的时间。

另外，slow log 保存在内存里面，读写速度非常快，因此你可以放心地使用它，不必担心因为开启 slow log 而损害 Redis 的速度。

设置 **SLOWLOG**

Slow log 的行为由两个配置参数(configuration parameter)指定，可以通过改写 redis.conf 文件或者用 `CONFIG GET` 和 `CONFIG SET` 命令对它们动态地进行修改。

第一个选项是 `slowlog-log-slower-than` ，它决定要对执行时间大于多少微秒(microsecond，1秒 = 1,000,000 微秒)的查询进行记录。

比如执行以下命令将让 slow log 记录所有查询时间大于等于 100 微秒的查询：

`CONFIG SET slowlog-log-slower-than 100`

而以下命令记录所有查询时间大于 1000 微秒的查询：

`CONFIG SET slowlog-log-slower-than 1000`

另一个选项是 `slowlog-max-len` ，它决定 slow log 最多能保存多少条日志， slow log 本身是一个 FIFO 队列，当队列大小超过 `slowlog-max-len` 时，最旧的一条日志将被删除，而最新的一条日志加入到 slow log ，以此类推。

以下命令让 slow log 最多保存 1000 条日志：

`CONFIG SET slowlog-max-len 1000`

使用 `CONFIG GET` 命令可以查询两个选项的当前值：

```
redis> CONFIG GET slowlog-log-slower-than
1) "slowlog-log-slower-than"
2) "1000"

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

查看 **slow log**

要查看 slow log ，可以使用 `SLOWLOG GET` 或者 `SLOWLOG GET number` 命令，前者打印所有 slow log ，最大长度取决于 `slowlog-max-len` 选项的值，而 `SLOWLOG GET number` 则只打印指定数量的日志。

最新的日志会最先被打印：

```
# 为测试需要，将 slowlog-log-slower-than 设成了 10 微秒

redis> SLOWLOG GET
1) 1) (integer) 12                      # 唯一性(unique)的日志标识符
   2) (integer) 1324097834              # 被记录命令的执行时间点，以 UN
   3) (integer) 16                      # 查询执行时间，以微秒为单位
   4) 1) "CONFIG"                       # 执行的命令，以数组的形式排列
      2) "GET"                          # 这里完整的命令是 CONFIG GET
      3) "slowlog-log-slower-than"

2) 1) (integer) 11
   2) (integer) 1324097825
   3) (integer) 42
   4) 1) "CONFIG"
      2) "GET"
      3) "*"

3) 1) (integer) 10
   2) (integer) 1324097820
   3) (integer) 11
   4) 1) "CONFIG"
      2) "GET"
      3) "slowlog-log-slower-than"

# ...
```

日志的唯一 id 只有在 Redis 服务器重启的时候才会重置，这样可以避免对日志的重复处理(比如你可能会想在每次发现新的慢查询时发邮件通知你)。

查看当前日志的数量

使用命令 `SLOWLOG LEN` 可以查看当前日志的数量。

请注意这个值和 `slower-max-len` 的区别，它们一个是当前日志的数量，一个是允许记录的最大日志的数量。

```
redis> SLOWLOG LEN
(integer) 14
```

清空日志

使用命令 `SLOWLOG RESET` 可以清空 slow log 。

```
redis> SLOWLOG LEN
(integer) 14

redis> SLOWLOG RESET
OK

redis> SLOWLOG LEN
(integer) 0
```

可用版本：

>= 2.2.12

时间复杂度：

O(1)

返回值：

取决于不同命令，返回不同的值。

SYNC

SYNC

用于复制功能(replication)的内部命令。

更多信息请参考 Redis 官网的 Replication 章节 。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

不明确

```
redis> SYNC
"REDIS0002\xfe\x00\x00\auser_id\xc0\x03\x00\anumbers\xc2\xf3\xe0\x(
(1.90s)
```

TIME

TIME

返回当前服务器时间。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

一个包含两个字符串的列表： 第一个字符串是当前时间(以 UNIX 时间戳格式表示)，而第二个字符串是当前这一秒钟已经逝去的微秒数。

```
redis> TIME
1) "1332395997"
2) "952581"
redis> TIME
1) "1332395997"
2) "953148"
```

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。